An Empirical Study of the Performance of Preprocessing and Lookahead Techniques for Solving Binary Constraint Satisfaction Problems

By

Zheying Jane Yang

A THESIS

Presented to the Faculty of
The Graduate College at the University of Nebraska
In Partial Fulfillment of Requirements
For the Degree of Master of Science

Major: Computer Science
Under the Supervision of Professor Berthe Y. Choueiry
Lincoln, NE

Version of July 28, 2003

An Empirical Study of the Performance of Preprocessing and Lookahead Techniques for Solving Binary Constraint Satisfaction Problems

Zheying Jane Yang

University of Nebraska-Lincoln, 2003

Advisor: Berthe Y. Choueiry

In recent years, controversies have developed in the Constraint Processing community concerning the relative performance of different and well-known preprocessing and look-ahead algorithms. The goal of our study is to empirically characterize the relative performance of the various combinations of preprocessing and look-ahead schemes as a function of constraint probability and tightness of the CSP.

As preprocessing techniques, we study arc-consistency (AC) and neighborhood inverse consistency (NIC). As for lookahead strategies, we focus on the mechanisms of forward checking (FC) [**?**] and maintaining arc consistency (MAC) [**?**]. We restrict our study to binary CSPs, which we solve with systematic backtrack search using least domain (LD) as a heuristic for dynamic variable ordering (DVO).

Our study and observations are based on empirical evaluations. To this end, we wrote a generator of solvable, random CSP instances. Varying constraint probability and thightness, we generated a full range of random instances. We solved these instances using various combinations of the above-listed pre-processing and lookahead techniques in order to determine the conditions under which each combination of algorithms excells. We tested five preprocessing algorithms and seven search algorithms. We compared the results in terms of the number of constraint checks performed and CPU time necessary for finding the first solution. Our implementation is written in Java.

Our results show that the popular beliefs regarding the relative performance of commonly used algorithms do not hold under all, or even most, conditions or types of CSPs, severed of these beliefs are that AC2001 outperforms AC3, and MAC outperforms FC.

Acknowledgments

I am greateful to Dr. Berthe Y. Choueiry who initiated me to this research field, then instructed and helped me to finish this thesis.

I am honored to have Dr. Peter Revesz and Dr. Sebastian Elbaum on my examination committee.

I have used extensive material from the master thesis of Amy Davis [**?**] as carefully indicated throughout this document. Dr. Xingzhong Li provided help in presenting experimental results.

I would like to thank my colleagues, the members of the Constraint Systems Laboratory, especially Catherine L. Anderson, Lin Xu, Daniel Buettner, Eric Moss, Hui Zou, and Yaling Zheng for all their friendship and support while I was working on this thesis. No matter what problems I encountered they were *always* available to help me out of the difficulty. I would like to express my special gratitude to Catherine L. Anderson for reading carefully this entire dissertation and for providing corrections that greatly improved the presentation.

I would like also to thank to Ms. Deborah Derrick, who went though my whole thesis to provide me feedback on English.

Finally, I am grateful to my family, my parents and my son David Li, and above all my companion and best friend, my husband Xingzhong Li.

All experiments were carried out on PrairieFire.unl.edu.

Dedication

This thesis is dedicated to my husband Xingzhong Li, who supported and encouraged me while I was working on this research and also throughout my career.

Contents

List of Figures

List of Tables

Chapter 1

Overview

1.1 Motivation

The study of constraint satisfaction problems (CSPs) [?; ?] is an important branch of research in Artificial Intelligence (AI). A large collection of problems in AI and other areas of Computer Science can be molded as CSPs. Constraint Processing is used in circuit design [?], graph coloring [?], scheduling [?; ?; ?; ?; ?; ?], resource allocation [?; ?; ?], temporal reasoning [?; ?; ?; ?; ?], planning [?; ?; ?] and also in database processing [?; ?; ?; ?].

A Constraint Satisfaction Problem (CSP), which we formally define in Section 2.1, is a combinatorial decision problem and is in general **NP**-complete. Dechter [?], classifies the techniques solving a CSP into two categories, namely *constraint preprocessing* (by consistency inference, or constraint propagation) and *search* (also called conditioning). A combination of preprocessing and search is typically used to find the first solution for a CSP or to determine that no solution exists.

- *Preprocessing:* Preprocessing algorithms are generally polynomial-time procedures. They transform the original constraint problem into an equivalent, tighter one by removing inconsistent combinations from the problem definition. This reduces the size of the search space, which in turn reduces the cost of finding the solution. The most basic consistency propagation mechanism is arc-consistency (AC). Since arc-consistency can be efficiently computed, it is almost always executed prior to search.

We compare the two well-known implementations of arc-consistency, namely AC3 [**?**; **?**] and AC2001 [**?**]. Another, relatively more recent consistency propagation mechanism is neighborhood-inverse-consistency (NIC) [**?**]. NIC performs stronger consistency checking than AC but has a higher computational cost. We compare the performance of these preprocessing algorithms while varying the constraint probability and tightness of the CSP.

- *Search:* Systematic backtrack search (BT) is the most common algorithm for solving CSPs. It explores the search space of partial solutions in a depth-first manner. The algorithm extends step by step a consistent combination, and backtracks on decisions to explore systematically other possibilities when a dead-end is encountered. BT search algorithm is in general an exponential-time procedure.

Look-ahead schemas intertwine constraint propagation with search in a manner to reduce the size of the remaining search space after each conditioning operation. Look-ahead schemas are known to enhance drastically the performance of search by recognizing dead-ends before they even occur in search. The most common look-ahead techniques are, in increasing order of propagation power, *forward checking* (FC) [**?**; **?**], *directed arc-consistency* (DAC) [**?**; **?**], and *maintaining arc-consistency* (MAC) [**?**].

The goal of this thesis is to expand on the results of other researchers in the comparison of these algorithms [**?**; **?**; **?**; **?**]. In particular, we propose to specify under what conditions these results hold. The results include:

1. The implementation of AC with AC2001 [**?**] outperforms its implementation with AC3 [**?**; **?**].

2. NIC is a worthwhile consistency propagation mechanism [**?**].

3. MAC [**?**], a full lookahead strategy, outperforms forward checking, a partial lookahead strategy [**?**].

4. The implementation of MAC with AC2001 outperforms its implementation with AC3.

It is common knowledge that the performance of search depends on the order in which we explore the partial consistent combinations. Such an order can at best be heuristic. In our study, we choose to use the *least domain* (LD) heuristic under *dynamic variable ordering* (DVO) for all the tested strategies.

1.2 Strategies tested

The combinations of strategies we tested were obtained by varying two parameters:

1. The preprocessing techniques by constraint propagation to reduce the size of CSPs prior to search without losing solutions.

2. The look-ahead which intertwines search with constraint propagation.

We studied:

- Five preprocessing algorithms, denoted **P***.

- Three search algorithms, denoted **S***. And,

- Seven of their combinations, denoted **P:***+**S:***.

1.2.1 Preprocessing algorithms

AC, as a mechanism, can be implemented using the AC3 or AC2001 algorithms.

NIC is a consistency-checking mechanism at the conceptual level that requires some search. This gives rise to the following questions:

- Which look-ahead strategy is to be used when enforcing NIC? We choose to test the effectiveness of NIC under the two look-ahead strategies FC and MAC.

- Which arc-consistency algorithm to use in MAC? For MAC, we test both AC3 and AC2001.

We uniformly use dynamic variable ordering (DVO) for all the NIC strategies tested.

The five preprocessing (**P**) algorithms are as follows:

1. **P:** AC3
2. **P:** AC2001
3. **P:** NIC-FC
4. **P:** NIC-MAC-AC3
5. **P:** NIC-MAC-AC2001

1.2.2 Search algorithms

We study search when intertwined with the following two look-ahead schemas: forward-checking (FC) and maintaining arc-consistency (MAC). This yields the following search strategies (**S**):

1. **S:** FC
2. **S:** MAC-AC3
3. **S:** MAC-AC2001

1.2.3 Combinations of preprocessing and search

Seven of the combinations of our four preprocessing techniques and three search strategies are worth examining:

1. **P:** AC3 + **S:** FC
2. **P:** AC3 + **S:** MAC-AC3
3. **P:** AC2001 + **S:** FC
4. **P:** AC2001 + **S:** MAC-AC2001

5. **P:** NIC-FC + **S:** FC

6. **P:** NIC-MAC-AC3 + **S:** MAC-AC3

7. **P:** NIC-MAC-AC2001 + **S:** MAC-AC2001

In fact, the combination of 5 preprocessing algorithms with 3 search algorithms yields 15 hybrid search algorithms. However, since our purpose of this study is to compare the performance of AC3 and AC2001, MAC and FC, we designed the hybrid search algorithms based on MAC with inside (preprocessing step) and outside (search step) with the same AC algorithms, such that, if AC3 is used as the preprocessing algorithm, than in the search MAC-AC3 is used. In this way, we can compare the performance of equivalent hybrid search algorithms both based on AC3 and AC2001. For comparing the performance of FC and MAC, we also designed hybrid search algorithms based on FC and MAC with inside (NIC preprocessing) and outside (search) with the same lookahead algorithm. For instance, if FC-DVO is used in NIC, than FC-DVO should be used in search, too, which yields a hybrid search algorithm **P:** NIC-FC + **S:**FC. Similarly, if MAC-AC2001 is used in NIC, then MAC-AC2001 should be used in search, which yields a hybrid search algorithm **P:** NIC-MAC-AC2001 + **S:** MAC-AC2001.

Thus we discard eight combinations, such that,
P: NIC-FC + **S:** MAC-AC3 (or MAC-AC2001);
P: NIC-MAC-AC3 (or NIC-MAC-AC2001) + **S:** NIC-FC;
P: NIC-MAC-AC3 + **S:** MAC-AC2001; and
P: NIC-MAC-AC2001 + **S:** MAC-AC3.

Figure 1.1 shows the algorithms tested in this study.

1.3 Empirical study

We developed a generator of random CSP instances that guarantees the existence of at least one solution. We examined two sets of problems. The first set has 50 variables, each of domain size of 10 and the second set has 30 variables with domain size of 20 (see

Preprocessing		**Search**
$\left\{\begin{array}{l}\texttt{AC3} \\ \texttt{AC2001} \\ \texttt{NIC} \times \left\{\begin{array}{l}\texttt{FC} \\ \texttt{MAC-AC3} \\ \texttt{MAC-AC2001}\end{array}\right\}\end{array}\right\}$	$\times$	$\left\{\begin{array}{l}\texttt{FC} \\ \texttt{MAC-AC3} \\ \texttt{MAC-AC2001}\end{array}\right\}$

Figure 1.1: *Search strategies tested. Since NIC requires search, we tested 3 different implementations for various lookahead strategies.*

Section 3.1). We varied constraint probability from 0.02 to 0.4 with a step of 10 or 20. For each probability value, we varied constraint tightness from 0.05 to 0.95 with a step of either 0.05 or 0.1. In our study, we generated over 6,000 problem instances. For each variation of parameters of the random CSP generator, we measured the number of constraint checks, the number of nodes visited, and the CPU time. We averaged the results over 30 instances, and computed the standard deviation per point.

1.4 Questions addressed

In this thesis we addressed the following questions:

1. Does AC2001 perform better than the AC3 as measured by CPU time and number of constraint checks?

   Answer: No. The experiments show that there is not a significant difference in the number of constraint checks between the two algorithms. Although AC2001 performs fewer constraint checks than AC3, it requires more CPU time than AC3 in our implementation, resulting in a more costly search in terms of time.

2. Is the cost of NIC comparable to that of the weaker constraint propagation mechanism AC?

   Answer: No. NIC always performs more constraint checks and spends more CPU time than any AC algorithm (i.e., AC3 and AC2001). However, we should keep in mind that NIC accomplishes more by way of constraint propagation than AC. Consequently, NIC always reduces the search space by a larger amount than any AC algorithm. It is a good preprocessing algorithm when the problem has low constraint probability. When probability is high, the cost of NIC becomes prohibitive.

3. How do the various preprocessing algorithms compare?

   Answer: For low values of constraint probability (i.e., sparse CSPs), NIC outperforms any arc-consistency algorithm (i.e., both AC3 and AC2001). Otherwise AC based algorithms are more efficient.

   - When probability is between 0.05 and 0.1, we advise to choose NIC in combination with FC-DVO.
   - When probability is greater than 0.1, we advise to choose AC3 in combination with FC.

4. How do the hybrid search algorithm compare?

   Answer: The best overall combination when the probability is between 0.05 and 0.1, is NIC-FC combined with FC, which is **P:** NIC-FC + **S:** FC. If the probability is larger than 0.1, we should choose **P:** AC3 + **S:** FC.

5. When does MAC outperform FC?

   Answer: When the constraint graph is sparse and the constraints are loose, MAC outperforms FC. However, in all other situations, FC outperforms MAC.

1.5 Summary of contributions

Our contributions can be summarized as follows:

1. We developed a generator of random binary CSPs that guarantees the existence of at least one solution.

2. We evaluated empirically the performance of seven combinations of preprocessing and search algorithms.

3. We qualified the relative performance of both MAC versus FC, and AC2001 versus AC3.

4. We developed a Java library containing a large number of algorithms for CSPs.

1.6 Outline of the thesis

In this thesis, we tested seven search algorithms obtained by using combining two preprocessing mechanisms (i.e., AC and NIC) and two look-ahead techniques (i.e., FC and MAC). These various combinations are obtained by using two different algorithms to implement AC (i.e., AC3 and AC2001).

This thesis is structured as follows. Chapter 2 reviews the key concepts of a CSP. Chapter 3 discusses the generator of random CSP instances, which can be controlled by four input parameters. Chapter 4 presents the results of our experimentations on preprocessing and analyzes them. Chapter 5 discusses the relative performance of the various combinations we developed. It analyzes in particular the relative performance of FC and MAC. Chapter 6 summarizes our investigations and discusses open questions for future research. Finally, Appendix A describes our implementation and Appendix B is a glossary of the most common technical terms used in the area.

Chapter 2

Background

In this chapter, we introduce the concept of CSP, and how to solve a CSP. Further, we use the n-*queen* problem as an example to show that *look-ahead* strategy does improve the performance of the traditional BT search. Finally, we review the related works by the researchers in this field, and introduce two 'look-ahead' algorithms.

2.1 Defining a CSP

A Constraint Satisfaction Problem (CSP) is defined by a set of variables each of which has a domain of values, which can be assigned to the variable, and a set of constraining conditions involving one or more variables. Depending on whether the variable domains are discrete or continuous, finite or infinite, different types of CSPs can be defined. In the content of our work, we focus on finite, discrete CSPs. We define a discrete, finite Constraint Satisfaction Problem as follows:

Definition 2.1.1. *A discrete, finite constraint satisfication problem is given as the following three sets:*

1. A finite set of n variables: $\mathcal{V} = \{V_1, V_2, \ldots, V_n\}$;

2. A finite set of Domains, $\mathcal{D}$, such that for each variable, V_i, associated with a domain D_i of possible values, such that $D_i \in \mathcal{D}$, each D_i is a set of discrete values.

3. A finite set of constraints, $\mathcal{C}$, restricting the values that variables can simultaneously take, such that $C_{i,j,\ldots,k}$ is a relation $C_{i,j,\ldots,k} \subseteq V_i \times V_j \times \cdots \times V_k$, defining allowed combinations. Each constraint in a CSP enumerates the combinations of variables and values that are permitted by that constraint [1]

A solution to a CSP is an assignment of a value to every variable from the domain of that variable in such a way that every constraint is satisfied. In a finite CSPs the values in the domain are finite and, as such, the set of all possible assignments is finite.

Our study is restricted to binary CSPs, where each constraint involves only two variables. Usually a non-binary CSP can be transformed into binary one [**?**] before it is solved. In this study we consider only binary constraints.

A CSP can be represented as a constraint graph. The variables of the CSP are the nodes of the graph and each constraint is an edge between the variable involved in the constraint. The constraint graph may be dense (large number of edges) or sparse (small number of edges).

For example in Figure 2.1, the constraint between X and Y allows Y to take either value c or d when X is assigned value b. Each constraint contains *variable-value pairs* (vvps), combinations of vvps are represented by *tuples*, for example $\{$< X, b> <Y, c>$\}$. A constraint is a list of such tuples. Any tuple that appears in the constraint is a permitted combination. An example of a binary CSP is shown in Figure 2.1. There are three variables, $V = \{X, Y, Z\}$. Each variable has domain size of 2, $D_x = \{a, b\}$, $D_y = \{c, d\}$, $D_z = \{e, f\}$. The constraints are C_{xy} = $\{$<b, c>, <b, d>$\}$, C_{yz} = $\{$<c, f>, <d, e>$\}$, C_{xz} = $\{$<a, e>, <b, f>$\}$.

Definition 2.1.2. In exhaustive systematic search in solving CSP problems, each possible combination of the variables is systematically generated and then tested to see if it satisfies all the constraints. Each combination that satisfies all the constraints is a solution. The size of the Cartesian product of all the variable domains $\mid D_{v_1} \mid \times \mid D_{v_2} \mid \times \ldots \times \mid D_{v_n} \mid$. is the

[1] It is possible to define constraints intensionally, such as $V_i > V_j$, $V_k < V_s$. In our random generated CSP, we only consider enumerated constraints.

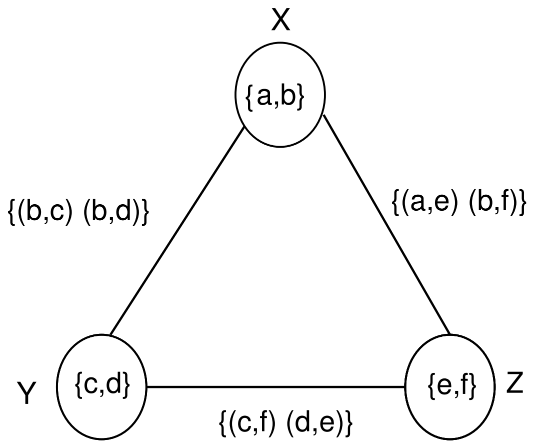


Figure 2.1: *A constraint graph of a binary CSP.*

size of the CSP. The size of the CSP depends on the number of variables, and the size of the variable domains.

The goal for solving a CSP can vary. It can be to find any solution (the first solution) or to find all solutions. our study is to find the first solution (single solution).

2.2 Strategies for solving a CSP

A CSP is generally **NP**-complete. It is solved with search. There are two primary search mechanisms for solving CSPs: systematic search and local search. Each mechanism employs different techniques to improve the overall performance. In this study, we study systematic backtrack search.

Systematic search exhaustively explores the search space, and finds either one or all solutions. Starting with the first variable, search assigns a value to it, checking constraints for satisfaction that apply to this *current variable*. If the constraints are satisfied, the next variable is assigned a value and again checked for satisfaction. These assigned variable-value pairs (vvp) are called a *partial solution*. The 'next' variable assigned a value 'extends' the partial solution. The variable currently being assigned a value is called the *current variable*. Unassigned variables are called *future variables*. Variables previously assigned values are called *past variables*.

All systematic search algorithms work with partial solutions, trying to extend these partial solutions until all variables in the search space are assigned. In simplest depth-first backtracking (BT) search, variables are instantiated sequentially in a predefined order. If an instantiation violates any of the constraints, the assigned value is discarded and the next value in the domain of the current variable is tested. If there are no values in the domain, backtracking is performed to the most recently instantiated variable that still has alternatives values available. The variable is reassigned and search continues, until finding the first solution or all solutions. If search backtracks to the first variable, and all the values in its domain have been discarded, then the problem has no solution.

BT does not record which combination of variable-value pairs (vvp) violate a constraints, such a combination is called *no-good*. It changes past variable values until it finds a value assignment compatible with the current assignment, or it finds that no solution exists. This search process suffers from *thrashing* [**?**], which keeps searching in different parts of the search space. For example, when BT search makes an assignment to V_i and this violates a constraint, search backtracks to the last variable successfully instantiated V_j, regardless if this is the source of the conflict or not. After restoring all domain values to D_i, a new assignment is made for V_j, and search proceeds. If V_j is not the source of conflict with V_i, then no assignment made to V_i will satisfy the constraint. The result is that all values in the domain of the variable V_j will be tried and discarded.

The performance of BT search can be improved with:

1. Look-ahead strategies, (such as FC and MAC)
2. Intelligent backtracking, (such as backjumping (BJ))
3. variable/value ordering heuristics

2.2.1 Look-ahead strageties

In recent years, many AI people study look-ahead techniques, these techniques are progress search algorithms based on BT search. We study these techniques in two levels, one is partial look-ahead, namely forward checking (FC) 2.2.2, another level is full look-ahead, namely maintaining arc-consistency (MAC) 2.2.3. In the following sections, we will mainly introduce these two look-ahead techniques.

'Look-ahead' checks forward from the current variable to the future variables to find inconsistencies in advance. It looks toward future variables at each new instantiation and revises the domains of some of the future variables to be consistent with the instantiation of the current variable. Which means it prunes the future variables domains by using *constraint propagation*. If the domain of any future variable has no remaining values (a.k.a, *domain wipe-out*), then backtrack occurs. Thus it detects 'dead-ends' earlier in the search tree.

2.2.2 FC algorithm

The forward checking algorithm was given in the paper "Increasing Tree Search Efficiency for Constraint Satisfaction Problems" by Haralick and Elliott) [**?**]. It is called forward checking because of its 'look-ahead' schema. Instead of performing arc-consistency to current variable, it performs a restricted form of arc-consistency to all future variables that are connected to the current variables via a constraint.

This restricted form of arc-consistency means that each time a variable V_i is assigned a value, the domain of future variables associated with V_i by a constraint is further restricted: Arc consistency between V_i and each V_j, where $j > i$, it will prune the search space if any variable domain is 'wipe-out'.

This pruning is achieved by the follow procedure: when a value is assigned to the current variable, any value in the domain of a future variable connected by a constraint to the current variable, that conflicts with the current assigned value is (temporarily) removed form the domains of the future variables. We say temporarily because if the domain of the any future variable becomes empty, the current instantiation is evaluated as inconsistent, upon which any value deleted during this instantiation will be put back into the domains of the future variables. FC continues to take the next value in the domain of the current variable for another assignment. If the domain of the current variable becomes empty, than backtrack occurs to the previously instantiated variable. A FC algorithm accomplishes the following:

- It makes partial lookahead toward the future variables each time, a new instantiation is made.
- It revises each future variables using the values in the current variable domain.
- It removes inconsistent values from the domains of all future variables connected to the current variable by constraint for one pass.
- Each time a variable is assigned a value, the domains of the future variables are restricted.
- If current domain is wiped out then backtrack occurs.

Figure 2.2 shows the flow-control of look-ahead technigues.

The current variable is the variable being instantiated. For instance, there is a consistent partial solution to $(i-1)$ variables, which means that all constraints involving these $(i-1)$ variables are satisfied. We call these $(i-1)$ variables $\{V_1, V_2, \ldots, V_{i-1}\}$ the past variables. The variable V_i is the current variable, and the variables $\{V_{i+1}, V_{i+2}, \ldots, V_N\}$ which have not yet been instantiated, are called future variables.

FC detects deadends higher up in the search tree. It allows branches of the search tree that will lead to failure to be pruned earlier than they would be in BT search.

In FC algorithm, which a solution is an assignment $\{V_1 \leftarrow v_a, \ldots, V_i \leftarrow v_b, \ldots, V_N \leftarrow v_z\}$, such that for all V_i, V_j such that $\{V_i \leftarrow v_x, V_j \leftarrow v_y\}$ is arc consistent.

In the main look-ahead procedure for both FC and MAC algorithm (shown in Figure 2.3), the current partial solution is $\{V_1, V_2, \ldots, V_i\}$. If there is a *domain-wipe-out* (DWO), which means that every value in the domain of some future variable is inconsistent with the assignment of the current variable, then there is no solution is exist in the

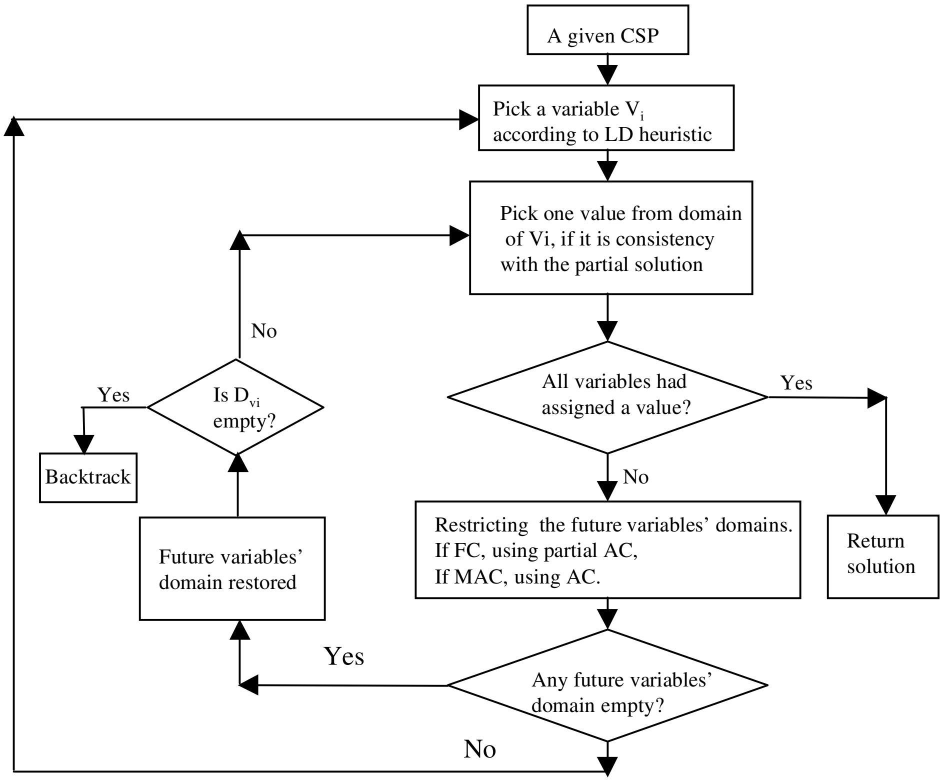


Figure 2.2: *The flow-control of look-ahead techniques.*

subtree of this current assignment, indicating that there is no extension for the current partial solution. If domain 'wipe-out' (DWO) is reached, then the domains of future variables are restored with the values deleted during that assignment. Note that, in Main Algorithm for FC and MAC, in procedure `SelectValue-X`, if it is used in forward checking, then X is replaced with 'partial-AC', otherwise, if X is replaced with 'full-AC', then it is used in MAC. In procedure `SelectValue-X`, DWO denotes domain wipe out.

If we combine preprocessing algorithm AC3, AC2001 and NIC with search algorithm FC with DVO, we have the following three hybrid search algorithms[2]:

1. AC3-FC

2. AC2001-FC

3. NIC-FC-FC

[2]Since every search algorithm is used DVO heuristic, DVO is omitted.

```
A main look-ahead procedure for both FC and MAC

Begin
while i =1 to N
        instantiate V_i ← SelectValue-X;
        if D_{V_i} is empty /* Domain wipe-out */
           i = i -1; /* backtrack */
           restore each Domain D_{V_j}, j > i;
        else
              i = i +1; /* step forward */
end
        if i=0
           Return 'inconsistent'
        else
              Return solution {V_1, ..., V_N}
End
```

Figure 2.3: *A main look-ahead procedure for both FC and MAC.*

2.2.3 MAC algorithms

Maintaining arc-consistency (MAC) [**?**] makes assignment for each variable in the same manner as FC with the different constraint checking continues until quiescence, as apposed to the single pass that FC makes. MAC checks the future variables against each other. MAC looks ahead with full arc-consistency. During search, MAC implements an arc-consistency algorithm. In our study, both AC3 and AC2001 are used within MAC.

MAC differs from FC in that, in main algorithm, it calls `SelectValue-full-AC` [3] Dynamic variable ordering (DVO) using LD can also be applied to the MAC algorithm, When it combines with AC3 as the full propagation algorithm, yielding the algorithm MAC-AC3. When MAC is combined with AC2001, we have MAC-AC2001 (see Chapter 1).

When an AC algorithm is combined with MAC-DVO, the following tasks are accom-

[3] In function `SelectValue-full-AC`, we can combine any AC algorithms. Both AC3 and AC2001 were implemented, instead of calling `SelectValue-partial-AC` shown in Figure 2.4. Since the pseudocode is samiliar with function `SelectValue-partial-AC` in FC, the only difference is that MAC performs full arc-consistency which does AC checks among the future variables. Thus we omit it.

```
Procedure SelectValue-partial-AC

Begin
while D_{V_i} is not empty
        pick one value v_a from D_{V_i}; and delete v_k from D_{V_i};
        DWO ← false;
                for all j, where j = i +1 to N
                     for each value v_b ∈ D_{V_j}
                          if (v_a, v_b) ∉ C_ij
                          delete v_b from D_{V_j};
                end for
                if D_{V_j} is empty
                   DWO ← true;
        if DWO = true
           Restore each D_{V_j}, where j = i +1 to N;
        else
               Return v_a
end while
Return false /* no consistent value */
End
```

Figure 2.4: *Subprocedure of look-ahead algorithm for* `FC`*, procedure* `SelectValue-partial-AC`.

plished during search:

- A variable is chosen for instantiation according to LD.
- For each instantiation, full-arc consistency on all future variables is performed.

If we combine these two look-ahead procedures with the follow preprocessing algorithms, AC3, AC2001, NIC-MAC-AC3, and NIC-MAC-AC2001, we have four hybrid algorithms:

1. AC3-MAC-AC3, denoted AC3-MAC3 in figures and tables;
2. AC2001-MAC-AC2001, denoteds AC2001-MAC2001 in figures and tables;
3. NIC-MAC-AC3-MAC-AC3, denoted NIC-MAC3-MAC3 in figures and tables;

4. NIC-MAC-AC2001-MAC-AC2001, denoted NIC-MAC2001-MAC2001 in figures and tables.

The various combinations of preprocessing and lookahead algorithms illustrated in Chapter 1, Figure 1.1.

2.2.4 Backtracking

During the search process, if conflict occurs, search goes back to the previous assigned variable. However, in intelligent backtracking (BJ) search backtracks to the variable that causes the conflict. This may avoid searching subtrees that have no solution.

Many intelligent backtracking search algorithms are developed, such as BJ [**?**], CBJ [**?**] that they learn while searching. Some of them can be combined with dynamic variable ordering heuristics to improve performance. In the present study, we do not address backtrack strategies.

2.2.5 Dynamic variable/value ordering heuristic

We define the order of instantiation as the order in which variables are assigned values.

In complete search algorithms, there are two types of variable ordering: *static* and *dynamic*. In static variable ordering, variables are sorted prior to search, according to the the heuristic chosen, such as least domain size. During search the next variable for instantiation is chosen according to this order. The order does not change during the search process. In dynamic variable ordering, the next variable for instantiation is chosen by a sorting process that is repeated after each successful instantiation. The aim of the DVO is to instantiate the variable whose assignment is most likely to cause a failure (dead-end). We call this the 'fail-first' principle (FFP), which is introduced by [**?**]. By using this principle, the 'dead-end' will be detected as early as possible in the search tree. Haralick and Elliott explain the FFP as: To succeed, try first where you are most likely to fail. They introduced and tested one of the best known DVO heuristic; choosing the variable with the least current domain as the next variable for instantiation (LD). The results of a FC search which combines with

DVO (LD) is given in the Figure 2.5. In the LD method, at each level of the search tree, the

DVO (LD) (*FutureVars*, *PastVars*):

Begin
next-var $\leftarrow$ nil
least-domain $\leftarrow 0$
For each variable V_i in *FutureVars*
  if V_i domain has the fewest values than *least-domain*
    best-var $\leftarrow V_i$
    least-domain $\leftarrow$ number of values in domain of V_i
return *best-var*
End

Figure 2.5: *Finding the next variable to expand using* `DVO with the LD`.

variable with the least current domain size (LD) is chosen for the next instantiation. This dynamic variable ordering heuristic (DVO) can be used to increase the performance of hard search algorithms.

The following example (see Figure 2.6) shows us that DVO (LD) has a large advantage over static variable ordering for this example figure. Let there be a CSP with three variables V_1, V_2, and V_3, with the following domains: D_{v_1}= $\{a, b, c, d\}$, D_{v_2} = $\{a, b\}$, and D_{v_3} = $\{a, b, c\}$. Let there be the constraints: $C_{v_1 v_2}$ = $\{(c, a)(c, b)\}$, $C_{v_2 v_3}$ = $\{(a, a)(b, c)\}$, and $C_{v_1 v_3}$ = $\{(c, a)\}$.

To compare the two search trees, one generated by using DVO (LD) and one using static ordering. It is clear that the tree generated by the DVO (LD) is much smaller than the one using static variable ordering with the first assignment V_1. The same heuristic used in static variable ordering can be employed in dynamic variable ordering, with the difference being that in dynamic variable ordering, the order of instantiation for all future or uninstantiated variables is re-evaluated after each instantiation.

There are many ways to implement FFP, such as instantiating a variable with minimal width ordering (MWO) [**?**], or minimal bandwidth ordering (MBO) [**?**], etc. In our study, we used the *least domain (LD)* size heuristic as the principal of dynamic variable ordering

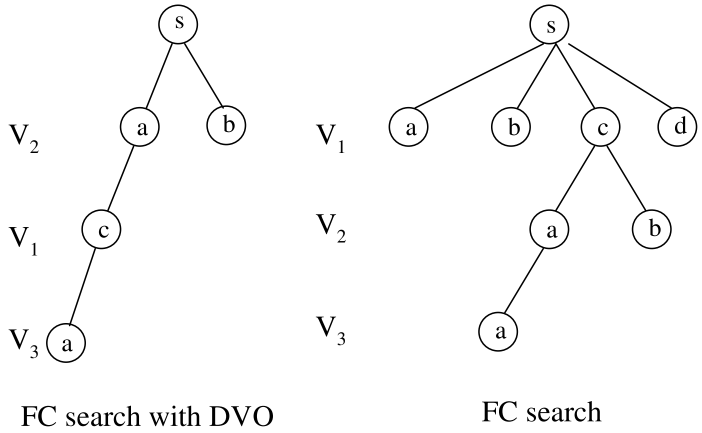


Figure 2.6: *Comparison of static ordering* `FC` *search and* `FC` *with* `DVO (LD).`

(see Section 2.2.5), This assignment order is according to increasing domain size. in which variables are sorted prior to search according to increasing domain size. During search, variables are instantiated in this static order. We can also use dynamic least domain heuristic, which at each level of the search tree, the variable with the smallest *remaining* domain is chosen for instantiation first.

As long as a variable is chosen for instantiation, every value of this variable is tested until find one which is consistent with the partial solution, then the search proceeds, next variable currently with least domain can be instantiated. When the domain of the current variable is empty, backtracking occurs.

2.3 Preprocessing using constraint consistency algorithms

As stated previous, a CSP is **NP**-complete may have an exponential number of possible assignments. In order to reduce this number of possible but inconsistent assignments, and hence improve the performance of search, preprocessing is used to filter the search space prior to starting the search process, resulting better run time search algorithms.

The general approach in preprocessing is consistency checking to achieve local consistency. There are many levels of local consistency, i.e., arc-consistency [**?**; **?**; **?**; **?**; **?**; **?**], path-consistency [**?**; **?**; **?**], k-consistency [**?**; **?**; **?**; **?**] and neighborhood-inverse-consistency (NIC) [**?**].

The appropriate level of local consistency to be chosen in preprocessing a constraint network is the central topic of this study. We studied two arc-consistency algorithms, AC3 and AC2001. Arc-consistency algorithms eliminate values that, at a local level, will not be a part of any solution.

The next level of consistency checking involves path-consistency. This ensures that any consistent solution for two variables is extensible to any third variable. Path-consistency is stronger than arc-consistency. Extending this further, k-consistency algorithms ensure that any locally consistent instantiation of $(k-1)$ variables can be extended to any k^{th} variable.

We also examined neighborhood-inverse-consistency (NIC), which combines inverse consistency and neighborhood consistency, resulting a greater reduction of search space than standard arc-consistency algorithms. NIC is a k-consistency algorithm, where the integer k represents the number of neighbors according to s given variable (see Chapter 4 for detail).

2.4 Search

We can combine consistency checking with BT search strategies to generate hybrid search algorithms. We need to consider what level of propagation is justified. Under the best conditions, the effect of performing arc or path consistency results in the search space being dramatically reduced. In some cases, it can entirely eliminate the need for search.

The complexity of consistency checks is determined by k, which denotes the level of consistency. At some point, given a CSP, higher orders of consistency checking become less effective than backtrack search. In practice, arc consistency is usually worth doing, because it is relatively cheap. However, for higher levels of consistency, because the complexity of enforcing k-consistency is exponential in k, there is a trade-off between the efforts spent on preprocessing and that spent on search. The comparison of the performance of AC and NIC is the focus of this study.

2.5 An illustrative example: the n-queen problem

Many problems can be cast to a finite CSPs, such as SAT [?; ?] on conjunctive normal form (CNF) formulae. It corresponds directly to a finite discrete CSP where all the domains contain only the boolean values $(0, 1)$, and each constraint contains exactly all the satisfying assignments of one particular clause.

In this section, we describe the n-Queen problem as an example of a CSP. The goal of the n-queen problem is to place n queens on an $n \times n$ chess-board so that no queens can attack each other. Because queens can move horizontally, vertically, and diagonally, this means that there can be only one queen per row and per column, and that no two queens can occupy the same diagonal. It is obviously that this problem has n finite variables, which are queens. The domain size is also finite, where $D_i = \{1, 2, \ldots, n\}$, $i = 1, 2, \ldots, n$, which i denoting the position of X_i in row i.

2.5.1 Model of the 4-queen problem as a CSP

In our model, each variable represents a column of the board. The domain of each variable represents the row of the board. The question is which row should each queen be placed, see Figure 2.7.

For this 4-queen problem, we have a set of four variables: V = $\{X_1, X_2, X_3, X_4\}$. The domain for each variable is the same: D =$\{1, 2, 3, 4\}$. We have also the set of constraints:

1. $X_i \neq X_j$ denotes that no two queens can be on the same row;
2. $\mid X_i - X_j \mid \neq \mid i - j \mid$ denotes that no two queens can be on the same diagonal.

We also write constraints in extension:

$C_{1,2}$ = {<1, 3> <1, 4> <2, 4> <3, 1> <4, 1> <4, 2>}

$C_{1,3}$ = {<1, 2> <1, 4> <2, 1> <2, 3> <3, 2> <3, 4> <4, 1> <4, 3>}

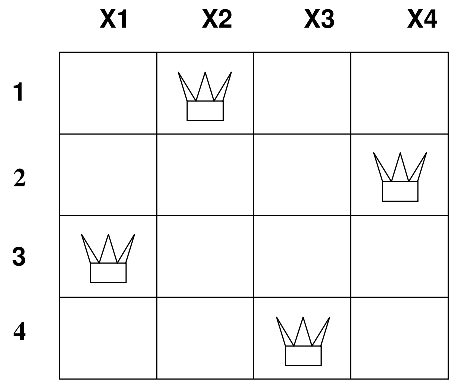


Figure 2.7: *4-queen problem as a CSP.*

$C_{1,4} = \{<1, 2> <1, 3> <2, 1> <2, 3> <2, 4> <3, 1> <3, 2> <3, 4> <4, 2> <4, 3>\}$

$C_{2,3} = \{<1, 3> <1, 4> <2, 4> <3, 1> <4, 1> <4, 2>\}$

$C_{2,4} = \{<1, 2> <1, 4> <2, 1> <2, 3> <3, 2> <3, 4> <4, 1> <4, 3>\}$

$C_{3,4} = \{<1, 3> <1, 4> <2, 4> <3, 1> <4, 1> <4, 2>\}$

2.5.2 Constraint graph for the 4-queen problem

Figure 2.8 shows the constraint graph for the 4-queen CSP. In the constraint graph each

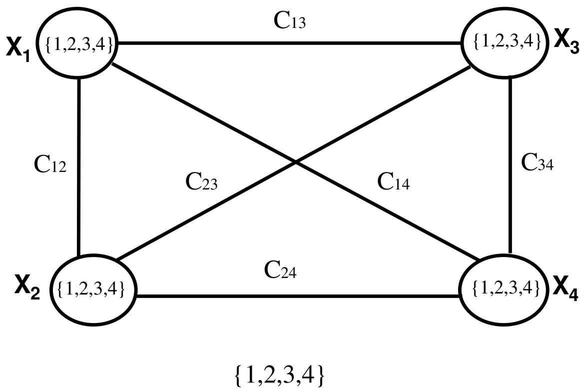


Figure 2.8: *The constraint graph of the 4-queen problem.*

node represents a variable of the CSP, and each edge represents a binary constraints.

2.5.3 Using preprocessing to solve the 4-queen problem

If we apply arc-consistency to the 4-queen problem, there is no effect, in that no values in the domains of the variables are eliminated. In this case, arc consistency is not the cost of effective.

However, if we apply path-consistency and NIC to 4-queen problem, the domains of the variables are reduced to the following: $D_1 = \{2,3\}$, $D_2 = \{1,4\}$, $D_3 = \{1,4\}$, and $D_4 = \{2,3\}$.

From this example, we see that preprocessing can reduce the search space if we apply the appropriate level of consistency. But if we apply too high a level of propagation to the CSP, the cost increase exponientially. if we apply too low a level of propagation, the search space is not reduced. In geneal, we can use a combination of low level of preprocessing algorithm and various search strategies to solve CSPs more efficiently.

2.5.4 Using search to solve 4-queen problem

By comparing the search trees for the 4-queen problem generated by BT search with those generated by using search with FC and MAC, we can see that BT search generates a much bigger search tree than those of both FC and MAC search. See Figure 2.9, Figure 2.10 and Figure 2.11.

In BT search, variables are instantiated sequentially. From Figure 2.9, we can see that BT search repeatedly explores the same subtree of a conflict causing the assignment to fail over and over again.

In FC search, each time a variable is instantiated, search checks the consistency between this current variable and all future variables connected with the current variables by constraints. Figure 2.10 shows that when the first variable X_1 is instantiated, since it has three neighbors, it checks the consistency between it and its three neighbors X_2, X_3 and X_4. As it turns out if variable X_1 is placed on position 1 then variable X_2 has two valid positions, positions 3 and position 4. However, if the partial solution is expanded to X_2=3, this will restrict X_3 and X_4, shch that there are no legal positions left, resulting

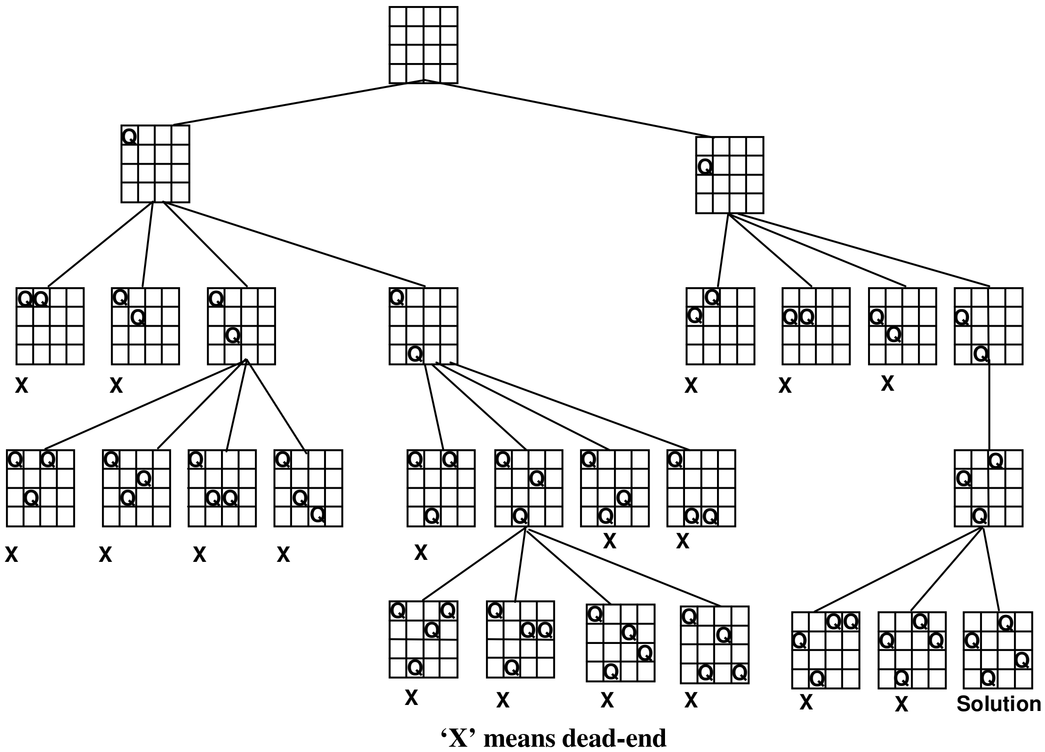


Figure 2.9: *BT search tree of the 4-queen problem.*

in the domains of variables X_3 and X_4 being 'wiped-outs'. FC search then backtracks to the previous instantiation X_1, picks another value from the domain of $D(X_1)$, which is the position 2, this leads to the solution $\{2, 4, 1, 3\}$.

FC ensures that each time a current variable is assigned a value, the domain of each future variable connected to this current variable via a constraint is revised to eliminate values that are inconsistent with the assignment of the current variable. This process is also called pruning. Because of pruning, subtrees without solution are cut off, and the *thrashing* is avoided. This is why FC search always visits fewer nodes than BT search [**?**]. Also FC assigns only values that are consistent with past variables. This means that all values in the future variables' domains are legal values since they are all consistent with the partial solution.

When using MAC with search, when the instantiation for variable X_1 is made, it will not only check the consistency between X_1 and three future variables X_2, X_3 and X_4, but

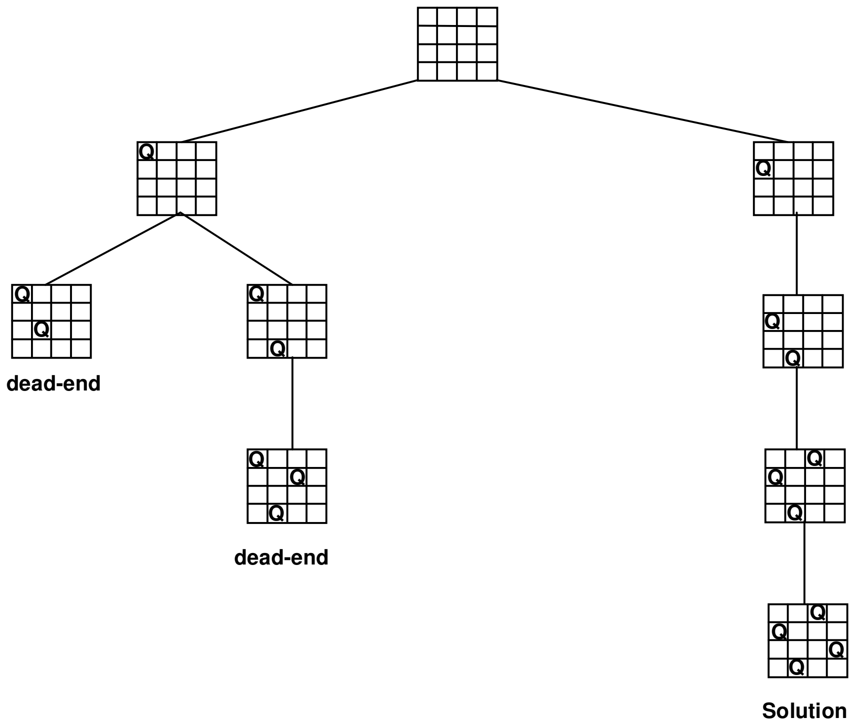


Figure 2.10: *FC search tree of the 4-queen problem.*

also check the consistency among the future variables X_2, X_3, and X_4. This means that for each instantiation, it does significant more constraint checks than FC does. Note that, it finds the solution only after two instantiations (see Figure 2.11 compared to the eight instantiations of FC.

Both solutions $\{2, 4, 1, 3\}$ and $\{3, 1, 4, 2\}$ for 4-queen problem are shown in Figure 2.12.

From this example, we can see that look-ahead algorithms do improve the performacne of BT search.

This 4-queen problem is small and easy to solve. If the problem is large, and if we combine preprocessing and search together, the search process will be much easier then using them separately.

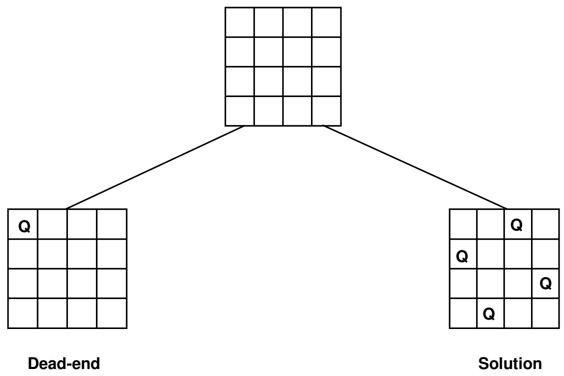


Figure 2.11: *MAC search tree of the 4-queen problem.*

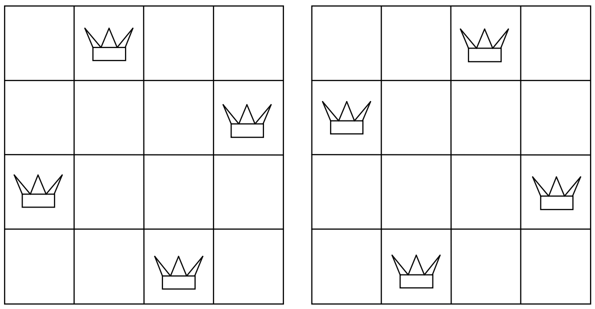


Figure 2.12: *Solutions of the 4-queen problem.*

2.6 Review of the related work

Bessière and Régin [**?**] introduced the arc-consistency algorithm AC2001, which is a modified version of the simple arc-consistency algorithm, AC3 (see section 4.1). They concluded that AC2001 is better than AC3 in terms of both CPU time and number of the constraint checks. Another preprocessing algorithm, Neighborhood-Inverse-Consistency (NIC) [**?**] (see section 4.2), was introduced by Freuder and Elfe. They concluded that NIC can achieve more pruning than the usual arc-consistency preprocessing algorithms. It is appropriate when memory is limited and the constraint network is sparse. It breaks a large

CSP into smaller ones, when the space requirement is at worst linear. Freuder introduced inverse consistency as $(1, k-1)$-consistency in [?]. Dechter [?; ?] introduced directed consistency. Dechter developed directed and adaptive consistency [?] algorithms, which could be considered as partial neighborhood consistency [?]. Specially, in [?], Dechter introduces various levels directed and adaptive consistency algorithms, which have much less cost than the equivalent consistency algorithms. Dechter and Meiri [?] had big contribution on comparing a variety of preprocessing algorithms experimentally. Further, in [?], Debruyne and Bessière studies various partial local consistency algorithms that are stronger than arc-consistency, without changing the structure of the network. In [?]. xxxx- also, add a discussion about Prosser's APES Tech Report and Stuart Grant thesis.

Two look-ahead search strategies, forward checking (FC) and maintaining arc consistency (MAC), are employed successfully by the constraint processing community. However, there is still a controversy over which algorithm, partial look-ahead provided by FC or full look-ahead provided by MAC is the most effective. In [?], Haralick and Elliott tested look-ahead algorithms using the n-queen problem and some random CSPs. The results showed that FC performed the best with the fewest constraint checks on those problems. They also concluded that higher level of look-ahead algorithm yields smaller search trees. Further, the effection of dynamic variable ordering also be tested, it is shown that DVO can improve performance of look-ahead techniques even further.

A detailed study of different look-ahead techniques for solving CSPs was reported by Nadel [?]. In this paper, various arc consistency algorithms were introduced, which combined with BT search to yield algorithms with various levels of look-ahead. These algorithms were also tested by empirical evaluation. The result of the study favored FC, which proved to be the most efficient algorithms. Nadel concluded that although higher level of look-ahead reduced the number of nodes visited, the extra work required at each node cancels out any savings. This result seems to contradict the report by Sabin and Freuder [?]. They compared the performance of AC3-based MAC and MAC-CBJ algorithms [?]. Their results were based on an average 50 instances of a randomly generated CSPs. The results shows that MAC performed much better than FC on hard and sparse CSPs. They concluded that FC is too weak to be effective on hard problems. They maintain that the pruning of MAC outweighs its overhead. Bessière and Régin [?] did their research by combining FC and MAC with various DVO heuristics, such as least domain, least degree, least size of domain plus degree. They also proposed a new DVO method, the minimum of the ratio of domain size over degree. They showed that CBJ is incompatible to a search algorithm with combines look-ahead technique with dynamic variable ordering.

These controversy still exists, which indicated to us that further exploration in this area was needed.

Sabin and Freuder tested the hybrid search algorithm FC-CBJ-DMD, which is a combination of forward checking (FC) with conflict-directed backjumping (CBJ) together with dynamic variable ordering (DVO) with least domain size ordering (LD) [?]. In [?], Prosser introduced many hybrid search algorithms, such as BJ, BM, and CBJ for the constraint satisfaction problems.

2.7 Constraint tightness and the phase transition behavior of CSPs

In 1991, Cheeseman [?] and other researchers found by empirical study that **NP**-complete problems exhibit a phase transition phenomenon. They found that the location of the phase transition and the steepness of the curve around this transition increases according to the size of the problem. For **NP**-complete problems (such as CSPs), we can find and choose an *ordered parameters* to describe the problem at a macroscopic level. These parameters are called critical values. A phase transition position is detected around the critical value of the ordered parameter. In CSPs, this critical value has been found to correlate to the constraint tightness, indicating that the position of a phase transition depends on constraint constraint tightness and also constraint probability [?]. This is illustrated in Figure 2.13. In this graph, along the x-axis, at the peak of the phase transition area, the order parameter takes on the critical value. On the left side of the critical value, there exists a large number of solutions, thus the value of the constraint probability is relative high, However, on the right side of the critical value, there exists few number of solutions, thus the value of the constraint probability is relative low. When the constraint tightness or probability is increased, the cost of solving CSPs is getting higher and higher, until reaching the critical value, in which the cost is the highest. After this point, the cost of solving CSPs is getting lower and lower, until constraint tightness or probability equals 1. Thus on the both side of the critical value, finding solutions for CSPs is cheap. In contrast, the phase transition is around the critical value, in which finding solutions for CSPs is costly. Since solving

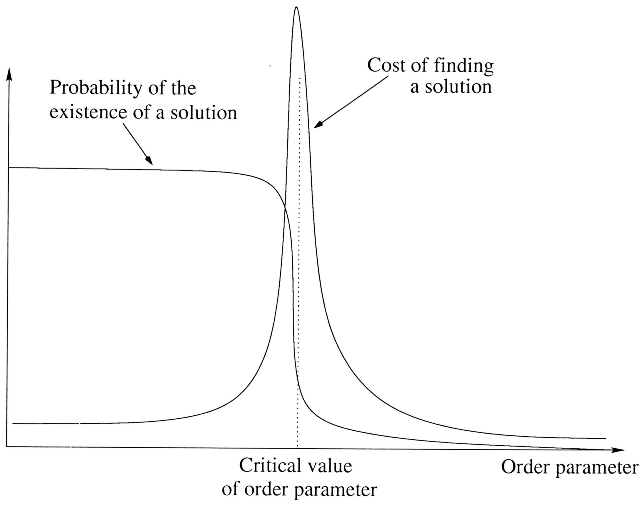


Figure 2.13: *The phase transition phenomenon of* **NP***-complete problems.*

CSP is generally an **NP**-complete problem, we expect the algorithms used FC and MAC to exhibit a phase-transition behavior. MAC applies a higher degree of look-ahead than FC does. In order to study the effects of using this increased look-ahead capability, we need to compare the performance of equivalent hybrid algorithms with MAC and FC over the same set of problems.

2.8 Summary

In this chapter, we introduced the concept of Constraint Satisfication problems and the strategies used to solve them. Originally BT search was used to solve CSPs. However BT search is not efficient, thus many techniques have been developed to improve its performance. Three such techniques are look-ahead techniques, backtracking and dynamic variable ordering heuristic. In this study, we introduced two look-ahead techniques (FC and MAC) and seven hybrid search algorithms baded on them. Furthermore we apply dynamic variable ordering (DVO) to the search procedures to save more constraint checks and CPU time.

For improving the BT search, before searching, we use constraint propagation technique (such as AC and NIC algorithms) to reduce the search space without loss of solutions. From this point of view, we developed seven hybrid search algorithms which combine preprocessing and search together, in order to compare relative performance between them.

Chapter 3

Methodology of experiments

We use randomly generated CSP models to measure the performance of the algorithms involved in this study. The alternative choice would have been to used real-world problems. However, real-world problems can not be controlled by explicit parameters in order to enable statistical analysis of the performance. Using randomly generated problems we can vary the characteristics of the CSP as needed. The experiments can be designed to monitor performance over a range of characteristics in the CSP as opposed to just over the general cases.

If we work on random problems, which allows us to create a large number of CSP instances with similar characteristics, we are able to categorize the performance of search on these problems. In order to do this, we first have to create a generator of random or instances. The generators available in the literature do not guarantee the existence of a solution, which would not suit our purpose. So, we developed one that would guarantee at least one solution.

3.1 Generator of random CSP instances

In this section we describe the detail of the generator, we designed. There are four general parameters in a random CSP generator, which are used to characterize the resulting CSPs. Using these four parameters, a large number of different CSP instances can be generated, all of which possess the same characteristics.

These four parameters are as follows:

1. n: the number of variables in the problem.

2. d: the number of values in the domain of each variable. In our implementation, all variables have the same domain size.

3. c: the number of constraints in a constraint network. An edge between a pair of nodes (variables) is counted as 1 constraint. This number must be greater than $(n-1)$ (to ensure connectivity), but is less than or equal to $n(n-1)/2$ (for a complete graph). In the analysis of the CSP, the number of constraints is translated to constraint probability (p). This is defined by: p=$\frac{C}{C_{max}}$, where C_{max} is the maximum possible constraints, $\frac{n\times(n-1)}{2}$.

   The constraints are chosen in a random manner. For example: if there are 10 variables, then the maximum number of possible constraints is $\frac{10\times 9}{2}$= 45, which occurs when p = 1. For the same number of variables, if there are 22 constraints, then the constraint probability p = 0.49. In our generator, the input parameter C is be an integer, where $(n-1) \leq C \leq n(n-1)/2$.

   Note that some researchers (e.g., [**?**; **?**]) in their randomly generated CSP, use constraint density instead of constraint probability. The constraint density is defined as: d = $\frac{e-e_{min}}{e_{max}-e_{min}}$, where e denotes the number of constraints in a constraint graph, e_{min} = $(n-1)$, and e_{max} = $n(n-1)/2$.

4. t: the tightness of each constraint. The constraint tightness is defined as number of forbidden variable value pairs in each constraint over all possible tuples. T = $\frac{disallowedtuples}{allpossibletuples}$. If the domain size is 10, then the maximum number of value pairs forbidden by a constraint is 10×10=100. In our generator, we chose to use an integer as the input parameter to avoid the ambiguity of rounding.

The random CSP generator provides virtually limitless supply of CSPs whose properties can be adjusted. The size of the CSP can also be adjusted to arbitrary levels. To successfully generate a CSP instance, we need to choose valid values for input parameters.

3.1.1 Guaranteed solution

Generally, random CSP generators do not guarantee the existence of a solution. This is the case for the CSP generator found in: `www.lirmm.fr/ bessiere/generator.html`.

A connected constraint graph requires a lower limit of $(n-1)$ on the number of edges (constraints). At the beginning of our investigations, we used the above referenced generator. However, it became apparent that in most of cases the instances generated did not have solutions. To avoid this problem, we implemented a modified random generator in Java, which we used in this empirical study.

There were two concerns in developing our random CSP generator. The first concern was to avoid generating reversed constraint. we define the reverse of a constraint as follows:

If a constraint between variable X_i and X_j is C_{ij}, the reverse constraint is C_{ji}, which should list the same allowed value pairs with the order reversed within each pair. C_{ji} is a redundant repetition of C_{ij} and hence can be dropped without loss of information. Because we assume that the constraint are symmetical. However, due to the random nature in which tuples are generated in a CSP generator, C_{ji} cannot be guaranteed to be equivalent to the reversed pairs of C_{ij}, and as such, they must be avoided.

The second concern in developing our CSP generator was to avoid duplicate tuples within a constraint. While duplicate tuples do not effect the solution, it would throw off the analysis with regards to the constraint tightness.

To avoid these two conditions, we use a function, `checkConstraint`, to make sure in constraint C_{ij}, $i < j$, therefor avoiding the conflict between the constraints C_{ij} and C_{ji} in a CSP instance. This function also avoids duplicated tuples appearing in the constraints.

To guarantee the existence of at least one solution, we use a function, `setSolution`, to randomly generate a solution. A second function, `adjustConstraints`, adjusts the tuples of the constraints to ensure that, in each constraint list, there is one tuple consistent with this solution. This does not affect the behaviors of the randomly generated instances, since the seed for the random CSP generator is the system time.

3.2 Experimental matrix

In our study, for each set of parameters, we fixed the number of variables to 50 and the domain size to 10, we varied constraint tightness T either from 0.1 to 0.9 with a step of 0.1, or from 0.05 to 0.95 with a step of 0.05. We also varied constraint probability from 0.024 (C=30, p=0.024), with a step of 10 or 20. Until 490 (C=490, p=0.4). We also computed some special points, such as p=0.1, where the number of constraint is 123, and p=0.2, where the number of constraint is 245, and so on. In this way, we can get the series of plots of constraint tightness vs. CPU time, number of node visited and number of constraint checks by varying constraint tightness or constraint probability, see Chapters 4 and 5 for detail.

For testing empirically the performance of algorithms, we ran each algorithm with 30 CSP instances of equal characteristics, calculating the average CPU time and number of constraint checks and number of node visited [**?**].

3.2.1 Number of constraint checks (`CC`)

We counted constraint checks in the following manner: When a value a for a variable V_i is checked for consistency with a value b for a variable V_j, this counts as one consistency check. If the variables involved are not constrained, then no check is counted. The consistency checking cost of search is the total number of constraint checks.

For each algorithm we used the same set of CSPs to see which one had the minimum number of constraint checks. This measure is implementation independent. Consistency checking is the main criterion for comparing the performance of the various algorithms.

3.2.2 CPU time

We measured the amount of CPU time consumed during search as the time to find the first solution. However, execution time is highly environment and implementation dependent, and is difficult to measure accurately. Thus we calculated the average case of 30 instances to try to get the result bias as small as possible.

Our experiments were carried out on machine–PrairieFire, which consists of 64 CPUs of 1.8GHz and 2GB RAM.

3.2.3 Number of nodes visited (NV)

This measure is implementation independent. Each time the algorithm makes an instantiation of a variable, it is counted as a node visited. The number of nodes visited increases with the number of backtracks during search. In this thesis, we also include the number of node visited for hybrid search algorithms. However, it does not reflect the effect of constraint propagation required for each instantiation. Thus it may not reflect the real cost of the search.

3.3 Statistical sampling

The experimental CSPs were generated from the given parameters N, D, C, T.

After generating a given instance of CSP, the instance was saved as a text file in the current directory where it can be accessed for the preprocessing and search. We wrote a Java implementation of preprocessing and search algorithms for this empirical study. The main programs for these are given in Appendix A and include five preprocessing and seven hybrid search algorithms. After the CSPs instances are generated with a given set of parameters, and then feed as inputs for each preprocessing and search algorithm and gives the results in CPU time and number of constraint checks.

For generating random instances of CSPs, we need to fix the number of variables in the constraint graphs, and the domain size of the variable.

We generated 30 instances of CSPs for each variation of constraint probability and tightness. Each instance, while having different solutions and constraints, has the same constraint probability and tightness. This allows us to average the constraint counts and

CPU time over the 30 instances and present this average as the representive of that specifically constraint probability and tightness.

The reason to choose 30 instances to make the average is that after computing the standard deviations of the CPU time for each of five preprocessing algorithms, and seven search algorithms, we found that because of the random behavior of the instances, the average of 30 instances is not getting worse than that of 100 instances, see Table 3.1.

Note that in Table 3.1, $Stdv$ denotes standard deviations. And the meaning of the names of the algorithms as follows:

- P_1 denotes **P:** AC3,
- P_2 denotes **P:** AC2001,
- P_3 denotes **P:** NIC-FC,
- P_4 denotes **P:** NIC-MAC-AC3,
- P_5 denotes **P:** NIC-MAC-AC2001,
- S_1 denotes **P:** AC3 + **S:** FC,
- S_2 denotes **P:** AC2001 + **S:** FC,
- S_3 denotes **P:** AC3 + **S:** MAC-AC3,
- S_4 denotes **P:** AC2001 + **S:** MAC-AC2001,
- S_5 denotes **P:** NIC-FC + **S:** FC,
- S_6 denotes **P:** NIC-MAC-AC3 + **S:** MAC-AC3,
- S_7 denotes **P:** NIC-MAC-AC2001 + **S:** MAC-AC2001,

3.4 Summary

In this chapter, we discussed the random generator for creating CSPs. There are four parameters for a random generator: N, the number of variables in constraint graph; D, the domain size of each variable; C, the number of constraints of a constraint graph. And T, the constraint tightness, the number of disallowed tuples inside each constraint. In the analysis, we represent C, the number of constraint to P, the constraint probability, and represent T, the constraint tightness, from number of disallowed tuples to the ratio of the number of disallowed tuples over total possible tuples.

If we change the combination of these four parameters, we can analyze the performance of each algorithm over a wide range of characteristics.

For generating connected CSPs, there are some restrictions in choosing parameters of generator, such as $n - 1 \leq C \leq \frac{n\times(n-1)}{2}$, and 0 $\leq T \leq D \times D$.

Table 3.1: *The comparison of standard deviations of CPU time from average the set of the different number of instances.*

Algorithms tested	CPU time (ms) for average 10 instances	Stdv of CPU time for 10 instances	CPU time (ms) for average 30 instances	Stdv of CPU time for 30 instances	CPU time (ms) for average 50 instances	Stdv of of CPU time for 50 instances	CPU time (ms) for average 100 instances	Stdv of of CPU time for 100 instances
P_1	77	27 (26.0%)	79	32 (28.8%)	71	31 (30.4%)	82	37 (31.1%)
P_2	94	33 (26.0%)	84	33 (28.2%)	94	36 (27.7%)	92	40 (30.3%)
P_3	588	52 (8.1%)	597	72 (10.8%)	582	53 (8.3%)	596	56 (8.6%)
p_4	759	97 (11.3%)	731	104 (12.5%)	725	111 (13.3%)	808	188 (18.9%)
P_5	766	88 (10.3%)	757	96 (11.3%)	745	129 (14.8%)	807	153 (15.9%)
S_1	110	27 (19.7%)	107	37 (25.7%)	89	37 (29.4%)	100	38 (27.5%)
S_2	116	38 (24.6%)	102	30 (22.7%)	109	35 (24.3%)	115	39 (25.3%)
S_3	740	103 (12.2%)	710	111 (13.5%)	707	153 (17.8%)	731	177 (19.5%)
S_4	875	125 (12.5%)	823	157 (16.0%)	825	187 (18.5%)	855	174 (16.9%)
S_5	605	44 (6.8%)	621	71 (10.3%)	599	61 (9.2%)	620	54 (8.0%)
S_6	1413	86 (5.7%)	1363	198 (12.6%)	1343	173 (11.4%)	1457	260 (15.1%)
S_7	1504	79 (5.6%)	1466	211 (12.6%)	1447	175 (10.8%)	1544	225 (12.7%)

Chapter 4

Empirical study of preprocessing algorithms

Consistency preprocessing methods are widely used to reduce the search space. Consistency preprocessing techniques were first introduced by Waltz [?] to improve the performance of image recognition programs. Consistency techniques can be used to eliminate inconsistent values for variables at shallow levels of the search tree, thus greatly reduce the search space. Preprocessing algorithms remove only local inconsistent values from domains of variables, and yield a tight and yet equivalent CSP.

Preprocessing algorithms can achieve local consistency. There are many levels of local consistency, such as arc-consistency (AC), path-consistency (PC), k-consistency, or neighborhood-inverse-consistency (NIC). The main focus of this study is to determine the level of local consistency that is appropriate for a given CSP. In this chapter, we compare the basic AC3 algorithm [?], its revised version AC2001 [?], and a stronger local consistency algorithm neighborhood-inverse-consistency (NIC) in which two popular lookahead searching algorithms are combined: maintaining arc Consistency (MAC) [?], and forward checking (FC) [?].

4.1 Arc consistency

In a constraint network, binary constraints correspond to an edge or 'arc' on a constraint graph, hence the name "arc consistency".

An edge, or arc, (V_i, V_j), is arc-consistent if for every value x in the domain V_i, there is some value y in the domain of V_j, such that the assignment $V_i \leftarrow x$ and $V_j \leftarrow y$ is allowed by the binary constraint between V_i and V_j. The concept of arc-consistency is directional, which means, if an arc (V_i, V_j) is consistent, then it does not necessarily mean that (V_j, V_i) is also consistent.

Arc-consistency eliminates values in the domain of each variable that can never satisfy a particular constraint (see Figure 4.1), and hence never be part of a solution. Arc consistency

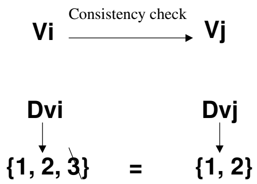


Figure 4.1: *Enforcing arc consistency.*

is achieved by continuing to delete any value from any domain D_i that fails this condition until no further changes occur in any domain (quiescence). It should be understood that deleting such values does not eliminate any solution of the original CSP.

The following algorithm, `Revise`, is an ancestor of present day arc-consistency algorithms, and is given in the Figure 4.2. The time complexity of `RRVISE` is $O(d^2)$, where d is the maximum domain size. Once `REVISE` deletes a value from a domain of some variable V_i, then each previously revised $arc(V_i, V_j)$ has to be revised again, because some of the values of the domain of V_j may have lost their support when a value was deleted from the domain of V_i. The arc-consistency algorithm AC1 [**?**] and AC3 [**?**] consider this

Procedure `REVISE(`V_i`, `V_j`)`

Begin
DELETE $\leftarrow false;$
For each x in D_i do
    if there is no such y in D_j, such that (x, y) is consistent,
    then
        Delete x from D_i;
        DELETE $\leftarrow true;$
    endif
endfor
Return *DELETE* **End**

Figure 4.2: *Algorithm* `REVISE`.

situation. If *DELETE* is true, then put the arcs that have been revised since last propagation back to the queue. We say a value a is $\in D_{V_i}$ is supported by a value $b \in V_j$, when $(a, b) \in C_{ij}$ hold.

AC1 is not efficient enough because once it revises one arc during an iteration, all arcs are processed again in the next iteration, regardless if the arcs are affected by the deleted values, or not. Next version of arc-consistency, AC3 improves AC1 by performing on only those arcs that were affected in last propagation. In AC3 algorithm, it picks one arc V_{km} from a Queue, then does `REVISE` on it, if it is revised, then put all constraints (arcs) which involved with V_k back to the end of the Queue, this process is called one iteration.

There are two versions of the AC3 algorithm, one is constraint-based and the other one is variable-based. In our study, we implemented the variable based AC3. The AC3 algorithm given by [**?**] which is constraint oriented, it stores the constraints to be propagated in a queue. The algorithm is given in Figure 4.3. The time complexity of constraint based AC3 is $O(n^2d^3)$, where n donates the number of nodes (variables) in the constraint graph, and d donates the maximum domain size of these variables. Note that in Figure 4.4, the function `REVISE-`x, where x denotes 3 or 2001.

The variable-oriented AC3 of [**?**] stores the variables in a queue of the variables. AC3 will terminate when either variable's domain is empty, or the CSP is arc-consistent. The

Procedure `AC3-Constraint-Oriented(`V_i`,` V_j`)`

Begin AC3
$Q \leftarrow (V_i, V_j) \in arc(G)$, *$i \neq j$;*
While not Q empty
    Select and Delete any $arc(V_k, V_m)$ from Q;
    if `REVISE(`V_k`,` V_m`)` **then**
    $Q \leftarrow Q \cup (V_i, V_k)$
    such that $(V_i, V_k) \in arc(G)$, *$i \neq k$, $i \neq m$;*
    endif
endwhile
End AC3

Figure 4.3: *Constraint-based* `AC3` *algorithm.*

Main AC algorithm

function `AC(in` X`:` *set*`): Boolean`
$Q \leftarrow \phi$;
for each $X_i \in X$ **do**
  for each X_j *such that* $C_{ij} \in C$ **do**
    if `REVISE-X(`X_i`,` X_j`,` $false$`)` **then**
    if *D(X_i) = ϕ* **then** *return* **false;**
    $Q \leftarrow Q \cup (X_i)$;
Return *Propagation-x(Q)*

Figure 4.4: *Main algorithm for variable-based* `AC3 and AC2001`.

pseudocode for variable-oriented AC3 algorithm is given in Figure 4.4 and Figure 4.5.

The time complexity of variable-based AC3 is $O(ed^3)$ where d represents the maximum domain size, e denotes number of edges in the constraint graph. The maximum number of edges possible, i.e., a complete graph is given as $e = n(n-1)/2$, $O(n^2)$. Thus, the worst case of time complexity is $O(n^2d^3)$, which is exactly the same as constraint-based AC3.

Note that, in Figure 4.5, function `REVISE-X` with $X = 3$ is the subprocedure for AC3, and with $X = 2001$ is the subprocedure for AC2001.

The main difference between AC2001 and AC3 occurs in the revise functions (i.e., `REVISE2001` and `REVISE3`. In `REVISE2001`, $D(X_j)$ holds the current domain of

Subprocedure for AC3 and AC2001

```
function Propagation(in Q: set): Boolean
Q ≠ φ;
take X_j from Q;
for each X_i ∈ X do
    for each X_i such that C_ij ∈ C do
        if REVISE-X(X_i, X_j) then
        if D(X_i) = φ then return false;
        Q ← Q ∪ (X_i);
Return true
```

Figure 4.5: *The psuedocode for the subpocedure of variable-based* AC3 or AC2001.

```
function REVISE3

function REVISE3(in X_i, X_j: variable): Boolean
CHANGE ← false;
for each v_i ∈ D(X_i) do
  if ∄ v_j ∈ D(X_j)/C_ij(v_i, v_j)
    then
    delete v_i from D(X_i);
    CHANGE ← true;
Return CHANGE
```

Figure 4.6: REVISE *procedure for* AC3.

X_j. We use the data structure Last(X_i, v_i, X_j) to store the value in $D(X_i)$ of the last support on a constraint or 'arc'. During the propagation, REVISE2001(X_i, X_j) looks for a support for a value v_i in $D(X_i)$ only if Last(X_i, v_i, X_j) is not in $D(X_j)$. Since if the support value for v_i is in $D(X_j)$ to ensure that v_i still have a support in D_j, which does not need to be checked for consistency again. If the value is not in $D(X_j)$, it must be deleted since last propagation, thus we need to check the current domain for a new support, starting from the index that is greater than d (the position of the last support value).

Time complexity of AC2001 is $O(ed^2)$, which an improvement over than AC3. However, for this algorithm, an extra data structure Last(X_i, v_i, X_j) is needed, which con-

```
function REVISE2001

function REVISE2001(in X_i, X_j: variable): Boolean
CHANGE ← false;
for each v_i ∈ D(X_i) do
    if Last(X_i, v_i, X_j) ∉ D(X_j) then
       if ∃ v_j ∈ D(X_j)/v_j >_d Last(X_i, v_i, X_j) ∧ C_ij(v_i, v_j)
       then Last(X_i, v_i, X_j) ← v_j;
       else
            delete v_i from D(X_i);
            CHANGE ← true;
Return CHANGE
```

Figure 4.7: REVISE *procedure for* AC2001.

sumes additional space. Thus the trade-off between time and space needs to be considered before using AC2001.

Bessière and Régin [?] claim that the data structure $Last(X_i, v_i, x_j)$ is light. However, in actuality, the data structure can grow to as much as $O(n^2d)$, with the need to store as many variable-value pairs. This could make the cost so large as to not be worth the added benefit.

4.2 NIC consistency

Neighborhood-Inverse-Consistency (NIC) [?] combines two ideas:

1. Neighborhood consistency
2. Inverse Consistency

Definition 4.2.1. A problem is (i, j)-consistent if any solution to a subproblem of i variables can be extended to a solution including any j additional variables.

Basically, most forms of consistency can be viewed as special cases of (i, j)-consistency [?]. When i is $(k-1)$ and j is 1 we have k-consistency [?]. If both i and j are 1 we have arc-consistency (AC) [?]. If i is 2 and j is 1, we have path-consistency (PC) [?]. When i is 1 and j is $(k-1)$ we have k inverse consistency.

Definition 4.2.2. The variables connected by an edge to a given variable V_i in a constraint graph are called the neighborhood of V_i.

Neighborhood-Inverse-Consistency (NIC) requires that every variable V_i be k-inverse consistency. In this application, k equals the number of variables that make up the neighborhood of V_i. The value of k changes with each variables, V_i where $i = 0 \to n$. In essence, NIC solves multiple subproblems that are made up of the three following sets:

- the set S that contains the variable V_i and those variables that make up its neighborhood,
- the set of domains of each variable D_{S_i}, and
- the set of edges that connect a variable in S_i with another variable in S_i.

NIC is stronger than any AC algorithm, since it deletes more inconsistent values. An example is given in Figure 4.8. There is no solution for the neighborhood of variable U if the

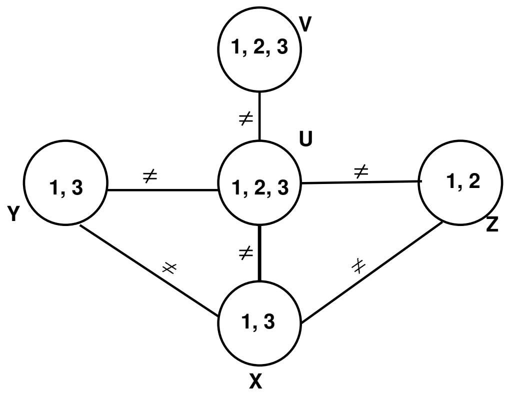


Figure 4.8: *The Example for NIC*

assignment $U = 1$. The NIC algorithm we used in this study was shown in Figure 4.9. All constraints in the constraint graph are inequalities.

Because NIC needs to test the solution with respect to each value in the subproblem, a search algorithm is contained within it. By applying 3 different lookahead algorithms inside

Neighborhood Inverse Consistency

function `NIC`
Insert each variable V into agenda A
While $A \neq \phi$ **do**
  pick a variable V from A
  $deleted\, leftarrow\, false$
  For each value a in $D(V)$ **do**
    if there is no solution to $Nbd(V)$ with V assigned a **then**
     $D(V) = D(V) \setminus \{a\}$ **and** $selete = true$
     if $D(V) = \phi$ **then return** ”domain is wiped-out”
    endif
  if $delete = true$ **then**
  for each u in $Nbd(V)$ **do** A=$A \cup \{u\}$
endwhile
return ”consistent”
end `NIC`

Figure 4.9: *Neighborhood Inverse Consistency (NIC) Algorithm.*

NIC, we construct three NIC preprocessing algorithms. NIC-FC (denoted 'P_3' in figures and tables, NIC-MAC-AC3 (denoted 'P_4' in figures and tables), and NIC-MAC-AC2001 (denoted 'P_5' in figures and tables).

At the beginning of NIC, all variables are stored in a queue. Then picking a variable V, the program samples each value of a variable V and checks whether it appears in any solution with the neighbors of V. When a value is not consistent with its neighborhood, then it is deleted. This deletion may remove the support of values in the domains of the neighbors of variable V, and so we must insert all neighbors of variable V back into the queue for further consistency checking.

Usually, the time complexity of k-inverse consistency is comparable to that of k-consistency, which is $O(n^k d^k)$. However, in terms of space complexity, inverse consistency has an advantage. In general, to achieve k-consistency requires that constraints involved in $(k-1)$ variables be stored. This space requirement is $O(d^{k-1})$, where d represents the domain size of the variable, V_i. As Freuder [**?**] mentioned that to achieve inverse consistency of a particular variable only requires specifically those values involved in sub-solutions. In worst case, the space requirement is only linear. This is the reason why NIC uses neighborhood inverse-consistency instead of neighborhood-consistency.

4.3 The results of the empirical study

The NIC algorithm needs to solve all local sub-CSPs consisting of each variable and its neighboring variables.

We compare the performance of AC3, AC2001, and NIC-FC or NIC-MAC algorithms empirically. In MAC, we use either AC3 or AC2001 as the constraint propagation algorithm. In both FC and MAC, we apply dynamic variable ordering (DVO) (Section 2.2.5) using the least domain (LD) (Section 2.2.5) heuristic.

4.3.1 The comparison of number of the constraint checks

The CSPs we tested have 50 variables with domain size of 10. We show the performance data for the CSPs that had constraint probabilities of 0.049, 0.1, and 0.2.

Figure 4.10 (right), Figure 4.11 (right) and Figure 4.12 (right) show that AC2001 has fewer consistency checks than AC3, but the difference is insignificant.

For example in Table 4.1, when constraint probability p = 0.049 and constraint tightness t = 0.8, AC3 and AC2001 have $5,249$ and 4,216 constraint checks, respectively. AC3 had 1033 (24.5%) more constraint checks than AC2001. Similarly, when p = 0.1, and constraint tightness t = 0.7, AC3 and AC2001 have 9740 and 7526 constraint checks, respectively. AC3 had 2214 (29.4%) more constraint checks than AC2001 (See Table 4.2).

We can also see from Table 4.1 that NIC always has more constraint checks than either AC algorithm. The percentage of the difference between AC3 and NIC-MAC-AC3 is significant, it is shown in Table 4.3. Note that, NIC-MAC3 represents NIC-MAC-AC3, similarly, NIC-MAC2001 represents NIC-MAC-AC2001.

Figure 4.13 (left) and Figure 4.13 (right) show that NIC reduces the size of the CSP significantly more than either AC algorithm. When the constraint probability increases, this difference is larger.

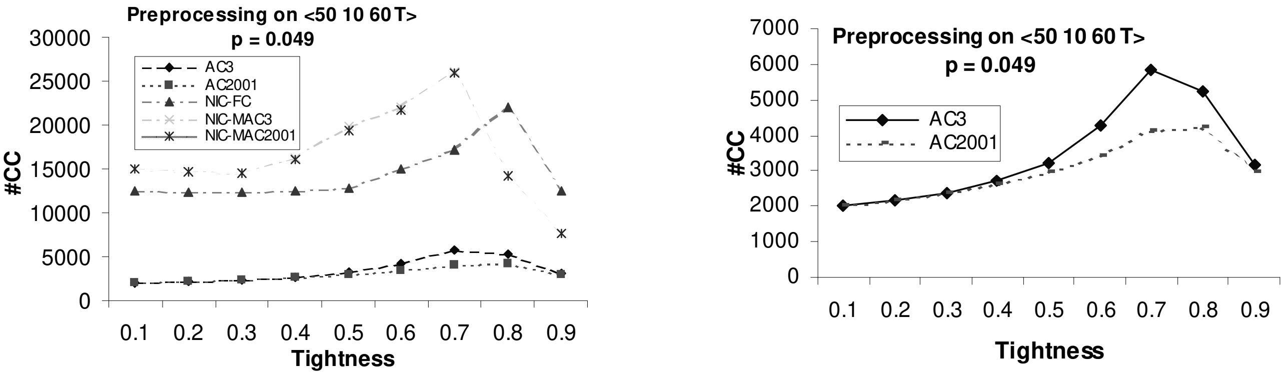


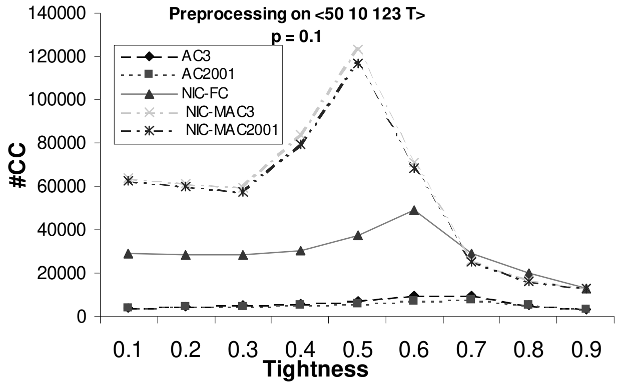
Figure 4.10: *Number of constraint checks for preprocessing for p = 0.049 (left). Comparing the number of constraint checks of AC3 and AC2001 for p = 0.049 (right).*

When the constraint probability is low (less than 0.05), NIC has no signifivant benefit over either AC algorithms, neither of which having much effect on reducing the search space. Figure 4.13 and show that when constraint probability is 0.082, along with a constraint tightness varing from 0.05 to 0.065, both AC algorithms algorithms and NIC algorithms do little to reduce the search space.

4.3.2 The comparison of CPU time

Figure 4.14 (right) and Figure 4.15 (right) show that AC3 consumes less CPU time than AC2001, but it is not significant. Table 4.4 shows this comparison, we see that when constraint tightness t = 0.7, AC3 takes 230ms, and AC2001 takes 287ms. The advantage of AC3 can be seen in both the phase-transition-area (Section 2.7) and non-phase-transition-area, regardless the value of constraint tightness. Similarly, when the constraint tightness p = 0.2, and the constraint tightness t = 0.6, it is at the peak of the phase transition, AC3 takes 512ms, AC2001 takes 692ms, the difference is 180ms, increased 35.2%.

NIC-based preprocessing algorithms spend significantly more CPU time than either AC-based algorithms. The larger the constraint probability, the larger the difference. For



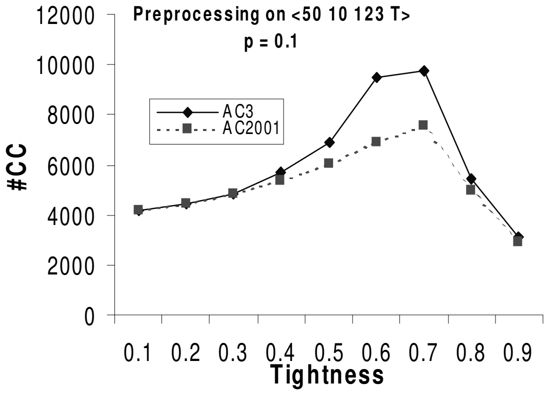
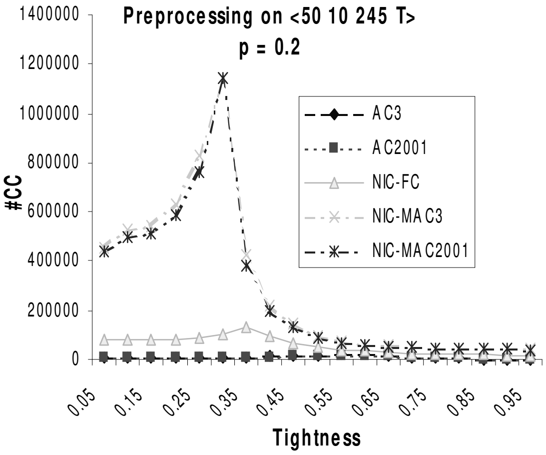


Figure 4.11: *Number of constraint checks for preprocessing for* p *= 0.1 (left). Comparing the number of constraint checks of AC3 and AC2001 for* p *= 0.1 (right).*

example, in Table 4.4, the constraint probability of 0.049, and constraint tightness t = 0.7 corresponds to the peak of the phase transition. From the table it can be seen that AC3 and AC2001 spend 93ms, and 116 ms of CPU time, but NIC-FC spends 545ms of CPU time, NIC-MAC-AC3 and NIC-MAC-AC2001 spend 735ms of and 660ms of CPU time, respectively. The difference between AC3 and NIC-MAC-AC3 is 2,085ms, which is a 7 fold increase. Similarly, the peak of the phase transition corresponds to a constraint probability of 0.2, and a constraint tightness t = 0.4, it can been seen that AC3 and AC2001 spend 297ms, and 430ms of CPU time, respectively, while NIC-FC spends 11,193ms, NIC-MAC-AC3 and NIC-MAC-AC2001 spend 12,946ms and 14,976ms of CPU time, respectively. The most significant difference is that NIC algorithm required more than 40 times the amount of CPU time that the corresponding AC algorithm. Table 4.3 shows the difference between AC3 and NIC-MAC-AC3.

Furthermore, Figure 4.14 (left) and Figure 4.15 (left) show that NIC-FC has better performance than either NIC-MAC-AC3 and NIC-MAC-AC2001 in term of CPU time.



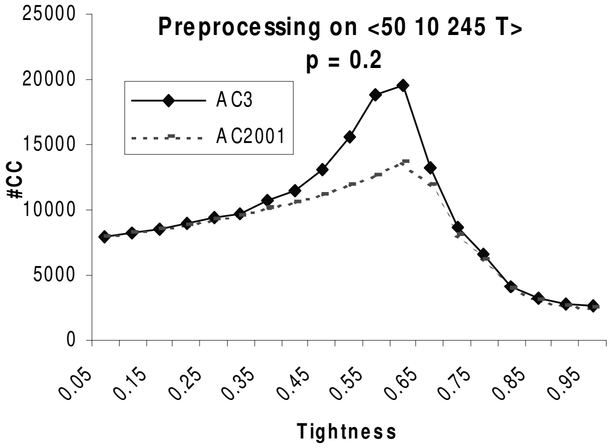


Figure 4.12: *Number of constraint checks for preprocessing for p = 0.2 (left). Comparing the number of constraint checks of AC3 and AC2001 for p = 0.2 (right).*

4.3.2.1 Varying constraint probability

If we vary constraint probabilities, and fix constraint tightness, we can find out when the constraint probability is higher than 0.4, the cost of NIC based preprocessing algorithms is too high. Thus we vary the constraint probabilities from 0.05 to 0.4.

Figure 4.16, and Figure 4.18 show that when the constraint probability is higher than 0.2, the costs are getting higher and higher for NIC based preprocessing algorithms. However, the costs of AC based preprocessing algorithms are very low (see Figure 4.16. From Table 4.5, we can compute the difference, when the constraint probability is 0.4, the costs of AC3 and NIC-MAC-AC3 are 720ms and 33370ms, respectively. The difference is 32650ms, NIC-MAC-AC3 costs 45 times than AC3.

4.4 Discussion

Based on the reports by Freuder and Sabin and Bessière and Régin, we expected that AC2001 would outperform AC3 in CPU time in the area of the phase transition. This expectation was based on the fact that in the algorithm AC2001, if support value 'mark'

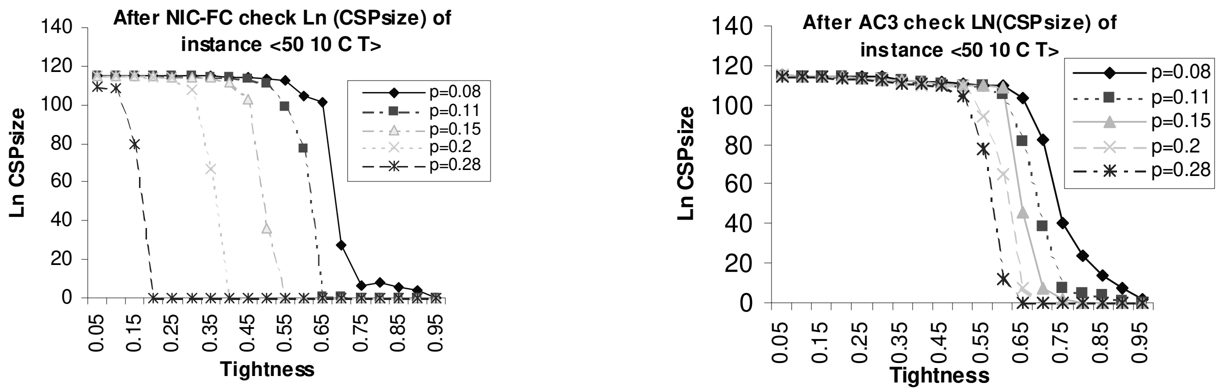


Figure 4.13: *Reduction of the size of the CSP (shown logarithmically), After preprocessing by NIC-FC (left). Reduction of the size of the CSP (shown logarithmically), After AC preprocessing by AC3 (right).*

is in the current domain then we don't need to check consistency, otherwise, it means the 'mark' has been removed by the last propagation, in which case we need to find another support value from the position bigger than d. However, this consideration appears to be bias, It is apparently based on the algorithm `REVISE`, when d is big that does not require a consistency check, and is valid when this is in the case for these condition, the size of the domain is larger, and constraint tightness is low. Because, when the constraint tightness is low, it is hard to find a support value 'mark' for an assignment, and if domain is large then jump step without checking the consistency might be also larger. However, if domain size is too large, then the time complexity is increased to $O(d^2)$, and it is possible that this saving does balance the extra cost. According to the experimental results, AC3 does outperform AC2001 on CPU time.

From the analysis above, the result did not surprise us, because from the constraint checks count, we can see that AC2001 does not significantly save constraint checks. This means that the savings in time could not balance the extra cost of working with an additional data structure. It does not seem to be a good trade off considering this added overhead.

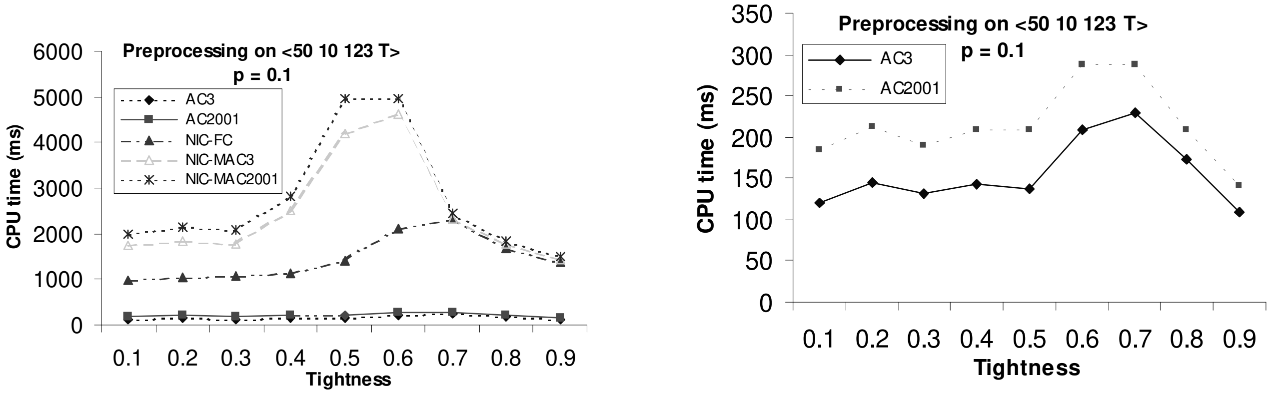


Figure 4.14: *CPU time [ms] for preprocessing for* p *= 0.1 (left). CPU time [ms] of AC3 and AC2001 for* p *= 0.1 (right).*

Thus when working with an AC algorithm in preprocessing, AC3 is the better choice over AC2001.

The conclusion of the paper [**?**] shows that AC2001 outperforms AC3 in both the number of the constraint checks and the CPU time, and that AC2001 has a significant advantage over AC3. Bessière and Régin maintained that AC2001 adds only a small data structure, `Last`, which stores the mark values since the last propagation. Actually this data structure can be quite large, up to $O(n^2d)$. In other words, if the CSP is large, the extra data-structure of AC2001 will become quite large, resulting in an overhead that is heavy to the point that the saved CPU time may not enough to balance the cost of the extra work. From this point of view, AC2001 can not significantly save CPU time over AC3 (see Table 4.2).

AC3 seems to be concise and simple, the results of from our experiments show that CPU time always favors AC3. Although AC2001 saved some consistency checks, this was insignificant. Thus we cannot assert that AC2001 is more efficient than AC3. The results of our experiments support our analysis that AC3 has the advantage over AC2001 in CPU time, and even though AC2001 has fewer consistency checks than AC3, the difference is insignificant.

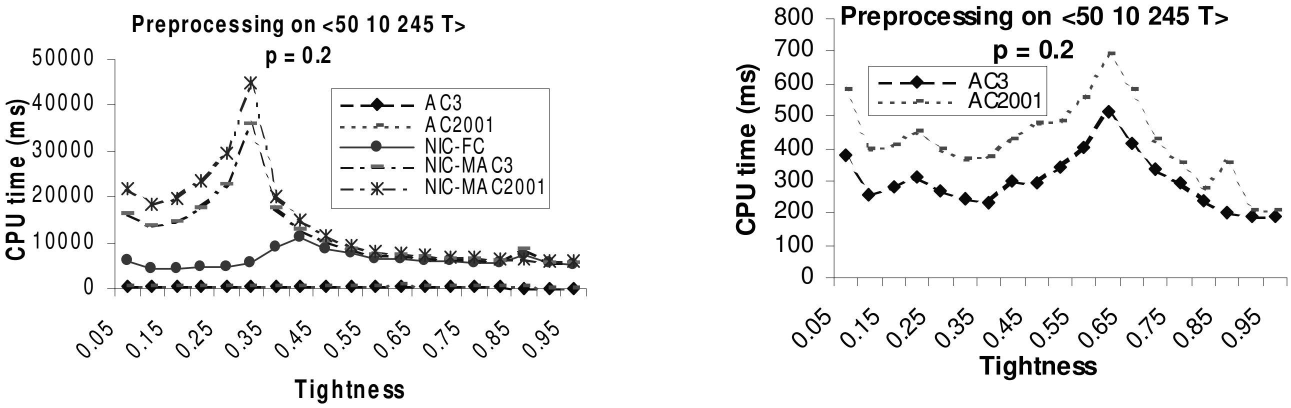
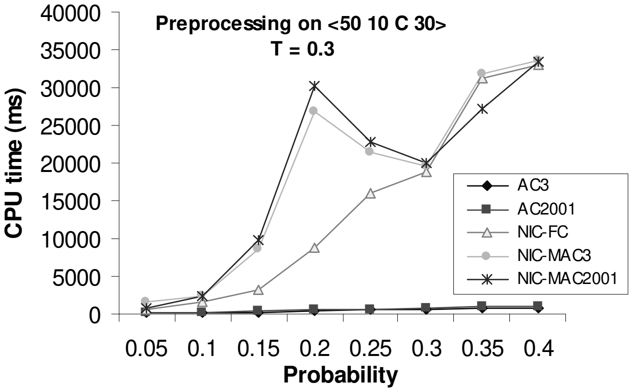


Figure 4.15: *CPU time [ms] for preprocessing for p = 0.2 (left). CPU time [ms] of AC3 and AC2001 for p = 0.2 (right).*

NIC is always more expensive than either AC algorithm. Since NIC does much more work than either AC3 or AC2001 during preprocessing, in that it enforces consistency by searching for local solutions for each value of the variables, this process is costly. However, we excepted that when the constraint probability is low, NIC will have an advantage over either AC algorithm. This expection is because the level of consistency it enforces is higher than either AC algorithms, which results in it removing more inconsistent values in each variable's domain. However, when constraint probability is high, the cost of checking each assignment in each subproblem is prohibitive. However, one should not compare AC and NIC based only on the results of preprocessing. NIC enforces a higher level of consistency than AC algorithms. To compare their performance, we need to compare the CPU time of the entire solving process, which is the topic of chapter 5.

Since the task of a preprocessing algorithm is reducing the search space and at a reasonable cost. NIC is too expensive to be used as a preprocessing algorithm. It is designed to have the advantage in space. NIC breaks a large problem into many small subproblems. Each time it processes a subproblem for a consistency check (get a local solution), it releases the memory it used for this subproblem before moving on to the next subproblem.



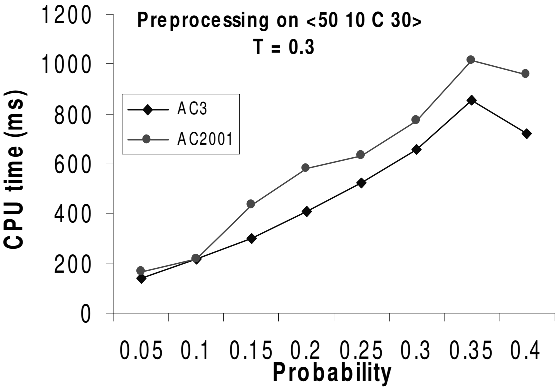


Figure 4.16: *CPU time [ms] for preprocessing for* t *= 0.3 (left). CPU time [ms] of AC3 and AC2001 for* t *= 0.3 (right).*

The NIC algorithm was introduced in 1996, when the amount of available average memory on an computer was low, and space issues were critical. Because computer technology has developed to the point where the concern over space is not as critical, and also we use depth-first search which is linear space, we consider more on time issue. Because NIC is slow, it is not a good consideration for a preprocessing algorithm comparing to the AC algorithms. We believe that the simpler and faster AC algorithm are more appropriate in practice. Our suggestion is when the constraint probability is low, we choose NIC preprocessing algorithm to reduce the search space. However, as a general rule, local consistencies should consume less time to detect that a branch of the search tree does not contain any solution than a search algorithm to explore this branch [**?**].

In [**?**], Sabin and Freuder concluded that NIC is more efficient than AC algorithms based on experimental results using instances of CSPs having extreme low constraint probabilities and constraint tightness (tightness $t = 0.4$, or $t = 0.5$ and p= 0.015, . . ., 0.0425. The authors believed that these characteristics constitute 'hard' CSPs. From our results, if for p $\leq$ 0.05, AC algorithms are not effective in reducing the search space. For this reason, NIC is appropriate. Because NIC outperforms AC algorithms in terms of filtering. However,

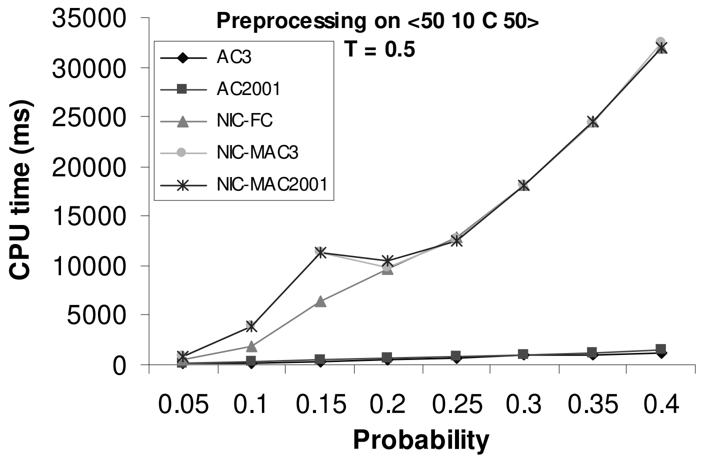


Figure 4.17: *CPU time [ms] for preprocessing for t = 0.5.*

to suggest the NIC outperforms AC on all instances of CSPs is not accurate. Furthermore, their results were obtained from a population of instances, 40% of which have no solutions.

In presenting our work, we have generated the plots representing the phase transition phenomeon of the NIC algorithm. The results illustrate that NIC has more constraint checks and spends more CPU time than either AC3 and AC2001. However, it does reduce the CSP's size significantly more than AC algorithms, resulting in less work during search.

While both AC and NIC are polynomial time algorithms, they still exhibit distinct phase transition. This demonstrates that the phase transition behavior (Section 2.7) is not a property confined to **NP**-complete problems.

4.5 Conclusions

NIC is good at filtering sparse CSPs where the number of constraints is small. However, when $p >$ 0.2, NIC becomes a costly preprocessing algorithm, as it does a large amount of extra work in finding local solutions.

AC2001 is not more efficient than AC3 in CPU time, even in the phase transition area.

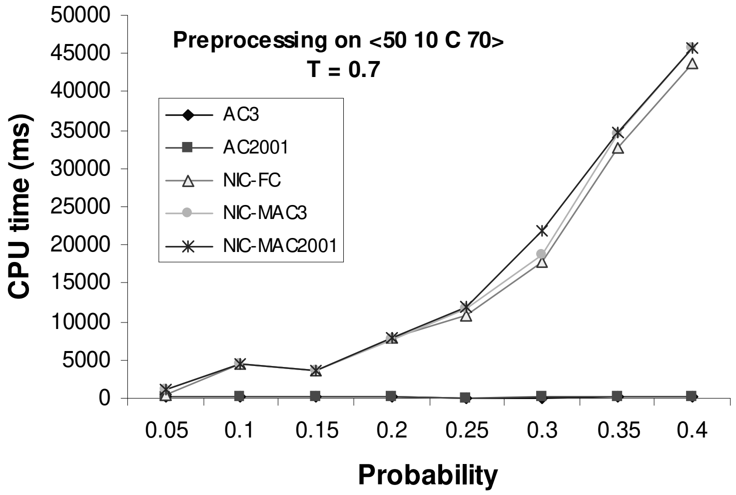


Figure 4.18: *CPU time [ms] for preprocessing for t = 0.7).*

However, it always has fewer constraint checks than AC3 does, though not significantly so.

After preprocessing by using AC2001 and AC3, a tightened CSP equivalent to the original one is obtained. In comparison, NIC does much more work than either arc-consistency algorithm AC3 or AC2001, resulting in much smaller CSPs. When $p < 0.1$, AC algorithms are not effective in reducing the search space.

NIC always spends more CPU time and has more constraint checks than either AC3 and AC2001. The higher the constraint probability, the higher the cost of NIC. When $p > 0.2$, the cost of NIC is prohibitive. The comparison of NIC and AC preprocessing algorithms are given in Table 4.6. NIC always reducing the larger amount of the search space than AC.

4.6 Summary

In this chapter, we used empirical evaluation to study and compare the performance of preprocessing algorithms on randomly generated CSPs. We compared AC3, AC2001 and NIC algorithms. When the constraint probability is high, the cost of NIC is prohibitive, favoring AC algorithms as preprocessing algorithms. However when it is low $p < 0.1$, NIC is more effective than AC algorithms in reducing the search space, but its cost is higher than either AC algorithm. For $p < 0.05$, both AC and NIC preprocessing algorithms are ineffective, they lose their ability to reduce search space except when constraint tightness is high ($t > 0.7$).

Number of constraint checks					
p = **0.049**					
Constraint tightness	AC3	AC2001	NIC-FC	NIC-MAC-AC3	NIC-MAC-AC2001
0.1	2010	2005	12429	15022	14982
0.2	2160	2154	12407	14708	14634
0.3	2377	2358	12409	14572	14486
0.4	2725	2618	12513	16226	16078
0.5	3219	2959	12878	19817	19388
0.6	4285	3428	14966	21983	21643
0.7	5830	4110	17245	26076	25950
0.8	5249	4216	21972	14263	14240
0.9	3174	2958	12511	7625	7622
p = **0.1**					
0.1	4150	4150	28833	63618	62871
0.2	4449	4421	28401	61474	60194
0.3	4871	4808	28166	59265	57520
0.4	5705	5345	30196	83576	79402
0.5	6865	6044	37566	123339	116596
0.6	9454	6924	48830	70723	68296
0.7	9740	7526	29207	25692	25223
0.8	5413	4997	19930	16589	16444
0.9	3096	2944	12754	13065	12983
p = **0.2**					
0.05	7926	7905	78753	456477	439910
0.1	8286	8228	80425	526149	501863
0.15	8537	8474	78095	543883	512954
0.2	8965	8815	82779	628086	586772
0.25	9384	9204	89627	831189	760008
0.3	9766	9574	102671	1143693	1141887
0.35	10786	10088	130435	422658	384261
0.4	11538	10621	98282	219214	199662
0.45	13045	11226	63603	144333	132698
0.5	15650	11907	53091	97853	91388
0.55	18758	12708	35620	72264	68263
0.6	19532	13702	34318	60892	58081
0.65	13305	11899	28519	55186	52772
0.7	8623	8097	24926	49581	47758
0.75	6588	6241	23562	46255	44715
0.8	4155	3955	19833	44416	43087
0.85	3295	3147	18693	42644	41470
0.9	2752	2618	16915	42397	41367
0.95	2586	2453	16464	40832	40014

Table 4.1: *Number of constraint checks for preprocessing for p = 0.049, 0.1 and 0.2.*

Number of constraint checks			
AC algorithms	$P = 0.049$	$P = 0.1$	$P = 0.2$
AC3	5249	9740	19532
AC2001	4216	7526	13702
Difference Percentage	1033 (24.5%)	2214 (29.4%)	5830 (42.5%)
Ratio	1.25	1.29	1.43
CPU time (ms)			
AC3	93	230	512
AC2001	116	287	692
Difference Percentage	23 (24.7%)	57 (24.8%)	180 (35.2%)
Ratio	0.8	0.8	0.74

Table 4.2: *Comparing of AC3 and AC2001 at the peak of phase transitions.*

Number of constraint checks			
Algorithms	$P = 0.049$	$P = 0.1$	$P = 0.2$
AC3	5249	9740	19532
NIC-MAC3	21972	25692	60892
Difference	16723	16336	41360
Ratio	0.24	0.38	0.32
CPU time (ms)			
AC3	93	230	512
NIC-MAC3	762	4619	12946
Difference	669	4389	12434
Ratio	0.12	0.05	0.04

Table 4.3: *Comparing of AC3 and NIC-MAC-AC3 at the peak of phase transitions.*

CPU time(ms)					
P = **0.049**					
Constraint tightness	AC3	AC2001	NIC-FC	NIC-MAC3	NIC-MAC2001
0.1	64	100	363	459	490
0.2	55	84	313	416	439
0.3	56	81	316	436	477
0.4	61	77	329	433	444
0.5	57	91	275	544	514
0.6	88	109	351	634	645
0.7	71	121	286	762	789
0.8	93	116	545	660	735
0.9	59	62	379	384	399
P = **0.1**					
0.1	121	184	994	1748	2005
0.2	144	213	1028	1852	2131
0.3	131	190	1057	1765	2088
0.4	143	209	1120	2518	2817
0.5	138	209	1407	4207	4955
0.6	208	287	2115	4619	4966
0.7	230	287	2335	2315	2449
0.8	173	209	1692	1768	1838
0.9	110	141	1373	1438	1505
p = **0.2**					
0.05	379	581	6041	16042	21866
0.1	259	396	4291	13492	18318
0.15	280	407	4356	14731	19601
0.2	312	450	4563	17336	23371
0.25	270	400	4783	22486	29566
0.3	247	366	5472	35899	45057
0.35	231	373	8980	17341	20196
0.4	297	430	11193	12946	14796
0.45	294	476	8548	10159	11490
0.5	339	482	7747	8709	9585
0.55	401	553	6623	7347	8161
0.6	512	692	6476	7224	7651
0.65	413	579	5939	6673	7207
0.7	333	427	5842	6484	6954
0.75	294	357	5729	6217	6750
0.8	238	274	5390	5963	6321
0.85	204	353	7381	8680	6483
0.9	192	208	5390	5689	6195
0.95	189	209	5299	5604	6107

Table 4.4: *CPU time [ms] for preprocessing for p = 0.049, 0.1 and 0.2.*

CPU time (ms)					
t = **0.3**					
Constraint probability	AC3	AC2001	NIC-FC	NIC-MAC3	NIC-MAC2001
0.05	140	165	605	1650	770
0.1	220	220	1590	2475	2360
0.15	300	435	3300	8540	9810
0.2	410	580	8765	26865	30260
0.25	525	630	16090	21370	22850
0.3	660	770	18810	19665	20020
0.35	855	1015	31225	31770	27110
0.4	720	960	32985	33530	33370
t = **0.5**					
0.05	140	165	490	765	850
0.1	220	300	1785	3845	3845
0.15	325	435	6345	11370	11370
0.2	520	605	9695	9725	10490
0.25	745	765	12880	12630	12580
0.3	960	1020	18120	18020	18100
0.35	1015	1150	24525	24360	24445
0.4	1265	1460	31970	32400	31970
t = **0.7**					
0.05	195	250	520	1125	1180
0.1	220	265	4615	4550	4610
0.15	165	165	3660	3615	3610
0.2	135	140	7880	7770	7800
0.25	110	110	10835	11780	11840
0.3	110	135	17855	18785	21750
0.35	120	145	32600	34385	34765
0.4	125	145	43670	45725	45615

Table 4.5: *CPU time [ms] for preprocessing for t = 0.3, 0.5 and 0.7.*

Comparing of the performance of AC and NIC				
Algorithms	$P < 0.05$	$0.05 \leq P < 0.1$	$0.1 \leq P < 0.2$	$P \geq 0.2$
AC	No	No	Yes	Yes
NIC	No	Yes	Yes	No

Table 4.6: *Comparing of the performance of* `AC` *and* `NIC`.

Chapter 5

Empirical study of search algorithms

In this chapter we compare empirically the performance of seven hybrid search algorithms obtained by mixing the 5 preprocessing techniques of Chapter 4, FC and MAC. We will demonstrate where one algorithm outperforms the others as a function of constraint probability and constraint tightness.

5.1 Results of our empirical study

This study was conducted to establish empirically the relative behavior of MAC and FC hybrid algorithms over a large population of CSPs. Our study is based on the CSPs obtained by varying t from 0.05 to 0.95 with a step of 0.05 or rarying t from 0.1 to 0.9 with a step of 0.1. For each set of CSPs, we vary the constraint number C from 40 ($p = 40/C_{max}$), to 50, 60,..., until 490 (p=0.4). We also compute some special points, e.g., p=0.1, , 0.15, 0.2, etc. The number of variables n and the domain size d were holded at 50, 10, respectively.

In this empirical study, for each point in the figures, we computed the mean of 30 CSPs obtained from the same set of parameters for random CSP generator (see Chapter 3).

5.1.1 Comparing the number of constraint checks

According to the results, we observed that, when the value of constraint probability is low ($p < 0.1$), the hybrid algorithms based on NIC perform better than the equivalent algorithms based on either AC3 or AC2001 (see Table 5.1 and Figure 5.1). This can be

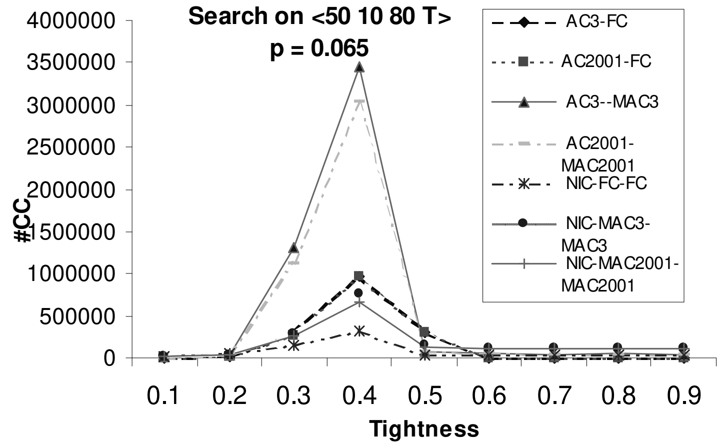


Figure 5.1: *Number of constraint checks for hybrid search for p=0.065.*

easily explained: NIC reduces the search space to a higher degree during preprocessing than either AC algorithm, which in turn improves the performance of the search procedure.

When the constraint probability is higher than 0.1, the FC based algorithms combined with the AC preprocessing algorithm performed better than FC based algorithms combine with the NIC preprocessing algorithm. With the higher constraint probability, see Figures 5.2 and 5.3, NIC enforces a higher level of consistency checking during the search for local solutions and consequently does more work. NIC is prohibitive as the preprocessing algorithm combined with any hybrid search algorithms on CSPs with a high constraint probability.

Generally, FC-based hybrid algorithms performed better than the equivalent MAC based algorithms except in the area of high constraint tightness and low constraint probability (p =0.049, t =0.7). AC3-FC showed much better performance than AC3-MAC3. Similarly AC2001-FC showed better performance than AC2001-MAC2001. When 0.05 $< p <$ 0.15, NIC-FC-FC outperforms all MAC-based hybrid algorithms.

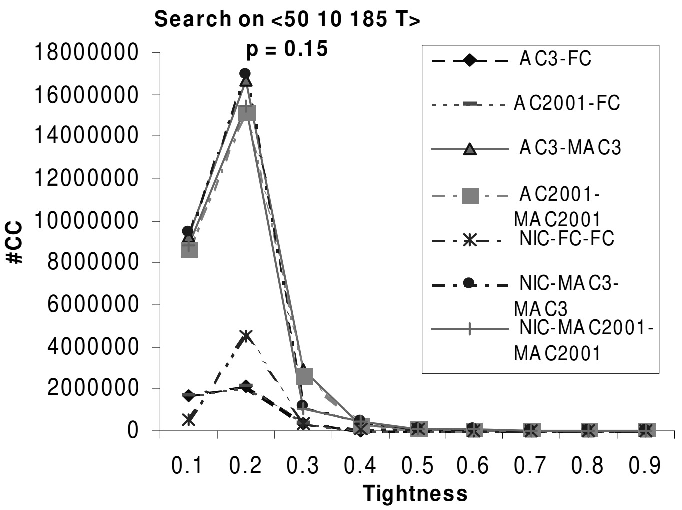


Figure 5.2: *Number of constraint checks for hybrid search for p=0.15.*

We also observed that the AC2001-based algorithms have fewer constraint checks than the AC3-based algorithms, but the difference is insignificant. For example, in the peak of the phase transition area, (see Table 5.1), AC3-FC has 974,817 constraint checks, while AC2001-FC has 974,430. The difference is 387, which is less than 1 percent.

When constraint probability is low (p = 0.049, shown in Figure 5.4), the algorithms with the best performance are NIC-MAC3-MAC3 and NIC-MAC2001-MAC2001. The AC3-FC and AC2001-FC hybrid algorithm have the worst performance among all seven hybrid search algorithms at this point. There are two reasons for this:

1. When constraint probability is low ($p < 0.05$), in sparse networks, AC is ineffective in reducing the search space, NIC has advantage over AC.

2. FC performs worse than MAC only when the constraint probability is low and constraint tightness is high. As noted by Davis [**?**] (see Figure 5.5), MAC performs better than FC on CSPs with high constraint tightness and low constraint probability.

Furthermore, in our study, no matter if constraint probability is low or high, the AC2001 based hybrid algorithms always had the limited benefit of fewer constraint checks over the

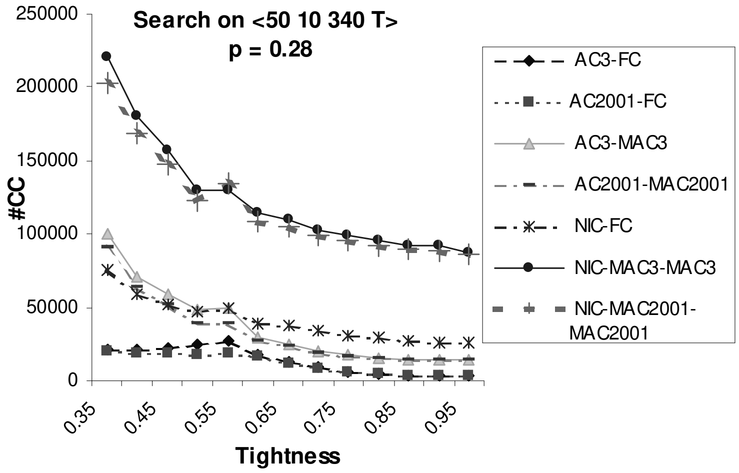


Figure 5.3: *Number of constraint checks for hybrid search for p=0.28.*

equivalent AC3 based hybrid algorithms, which confirms what Bessière and Régin claimed in [?]. When the constraint probability is higher than 0.05, p =0.065 and p=0.114, see Figure 5.1, the FC-based hybrid algorithms have better performance than the equivalent MAC based hybrid algorithm, this observation can be also seen in Figure 5.6, Figure 5.2), and Figure 5.3. However, the difference in the number of constraint checks is significant. For example, in Table 5.2, when p = 0.114, AC3-FC has 3,594,013 constraint checks, and AC3-MAC3 has 21,741,682 constraint checks, the different being 18,105,987, or 504%. When $p >$ 0.1 and $p <$ 0.15, see Figure 5.6, the FC-based hybrid search algorithms have better performance than any MAC-based hybrid algorithms. However, at these two points, NIC-FC-FC has the best performance among all seven hybrid algorithms (see Table 5.2, when p = 0.114 and Figure 5.6), thus we conclude that NIC as preprocessing algorithm is best choice on this condition.

In contrast to this, when p = 0.15 (see Table 5.1 and Figure 5.2), NIC-FC-FC no longer has the best performance. Both AC3-FC and AC2001-FC outperform it, due to the AC algorithms start reducing the search space to a larger degree.

Number of constraint checks							
p = 0.049							
	AC3		AC2001		NIC		
Constraint tightness	FC	MAC	FC	MAC	FC	MAC-AC3	MAC-AC2001
0.1	2562	30977	2556	30629	12981	43988	43605
0.2	2708	36675	2702	36365	12955	49222	48844
0.3	2914	34716	2894	34102	12949	46912	46231
0.4	3261	40701	3154	39227	13058	53614	52333
0.5	12392	50051	12133	44937	13425	61053	58402
0.6	55018	84390	54160	81008	26361	63808	60009
0.7	318190	118717	316470	104785	87214	82451	77987
0.8	5493	22005	4460	18939	22290	21390	21330
0.9	3272	7152	3056	6564	12589	10278	10275
p = 0.065							
0.1	6687	9503	6413	9150	28060	20929	20875
0.2	18670	40867	11907	31927	57725	43114	42664
0.3	276287	1312983	273571	1112109	153256	292614	267786
0.4	974817	3454415	974430	3034813	328120	750026	668055
0.5	295499	85168	295455	78850	40272	150425	140013
0.6	6060	54113	6021	51416	39163	113710	109717
0.7	5429	46851	5429	45405	39726	113484	111113
0.8	5115	47576	5115	47031	39356	105714	104913
0.9	4767	41969	4767	41799	40310	105796	105544
p = 0.15							
0.1	1648996	9306374	1648982	8617520	510020	9474920	8782006
0.2	2123527	1.66E+07	2123422	1.52E+07	4521114	1.69E+07	1.55E+07
0.3	318273	2917237	318184	2682762	385320	1128042	1033454
0.4	34766	269004	34235	241701	77244	440837	404105
0.5	15810	83310	13836	72861	72174	123911	115944
0.6	16853	40407	11392	32228	41016	57959	55456
0.7	9297	16347	8469	15022	24361	38883	37940
0.8	4901	10984	4651	10680	17010	32085	31499
0.9	3021	8979	2883	8837	15337	30774	30402

Table 5.1: *Number of constraint checks for hybrid search for p = 0.049, 0.065 and 0.15 .*

When the constraint probability is higher than 0.2 (p =0.28, see Figure 5.3 NIC based search algorithms perform poorly, due to the much larger number of constraint checks compared to the equivalent AC3 and AC2001-based hybrid algorithms. At this point, FC-based hybrid algorithms perform significantly better than MAC based algorithms. Figure 5.3 shows that, when $p < 0.2$, and $t<$ 0.5, the preprocessing algorithms are ineffective in reducing the search space. When 0.5 $<t<$0.65, the preprocessing algorithms begin to be effective at reducing search space, causing the cost of search to drop quickly. At the point $t > 0.65$, the search space is reduced to the point where search cost is low. During this, the peak of the phase transitions of NIC based FC and MAC algorithms is reduced to insignificance, which are invisible, but the phase transitions of AC based FC and MAC are

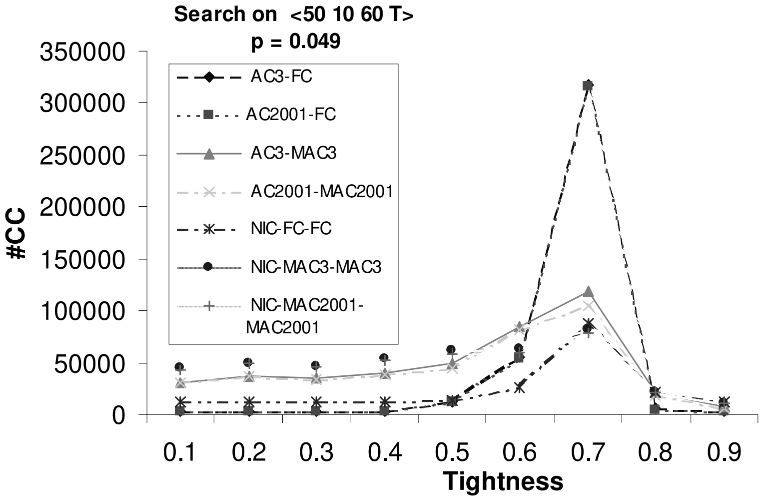


Figure 5.4: *Number of constraint checks for hybrid search for p=0.049.*

still apparent (see Figure 5.3). The comparison of the number of constraint checks between AC3-FC and AC3-MAC3, between AC3-FC and NIC-MAC3-MAC3, also between NIC-FC-FC and NIC-MAC3-MAC3 are given in Table 5.3.

Table 5.3 show that in most of cases FC based lookahead algorithms have less number of constraint checks than MAC based lookahead search algorithms, such that AC3-FC has always less number of constraint checks than AC3-MAC3. NIC-FC-FC has less number of constraint checks, when constraint probability is larger than 0.1. When the constraint probability is low, such that at the points of (p = 0.041, t =0.7), NIC-FC-FC has more constraint checks than NIC-MAC3-MAC3.

5.1.2 Comparing CPU time

In terms of CPU time, when $p<$ 0.1 (e.g., p=0.082, shown in Table 5.4, and Figure 5.7), all hybrid search algorithms based on AC3 have better performances than the equivalent hybrid search algorithms based on AC2001. For instance, AC3-FC has better performance than AC2001-FC. Similarly AC-MAC3 has better performance than AC2001-MAC2001

Probability \ Tightness	Low	High
Low	FC	MAC
High	FC	FC

Figure 5.5: *The performance regions distribution depends on various constraint probabilities and constraint tightness. MAC only wins in low constraint probability and high constraint tightness region.*

(see Figures 5.7 and 5.8).

The figures also show us that in the non-phase-transition area, NIC-based hybrid algorithms perform the worst. However, in the phase-transition area, NIC-based algorithms perform better than the equivalent AC based hybrid algorithms. For example, NIC-MAC3-MAC3 has better performance than AC3-MAC3. Similarly NIC-MAC2001-MAC2001 has better performance than AC2001-MAC2001.

AC3-FC has the best performance accross all values of tightness, followed by AC2001-FC. We can see $p<$ 0.1, NIC-FC-FC does not has the worst performance.

When $p > 0.1$ (e.g., p=0.14, see Table 5.4 and Figure 5.8), NIC-based hybrid algorithms perform worse than the equivalent AC-based algorithms. For instance, NIC-MAC3-MAC3 has worse performance than AC3-MAC3. Similarly NIC-MAC2001-MAC2001 has worse performance than AC2001-MAC21 over all. This result is as expected. The FC based hybrid algorithms have the best performance, e.g., AC2001-FC and AC3-FC.

The performance of AC3-FC is better than that of AC2001-FC in CPU time. For example, in Table 5.4, p = 0.14, t = 0.25, at the peak of the phase transition, AC3-FC and

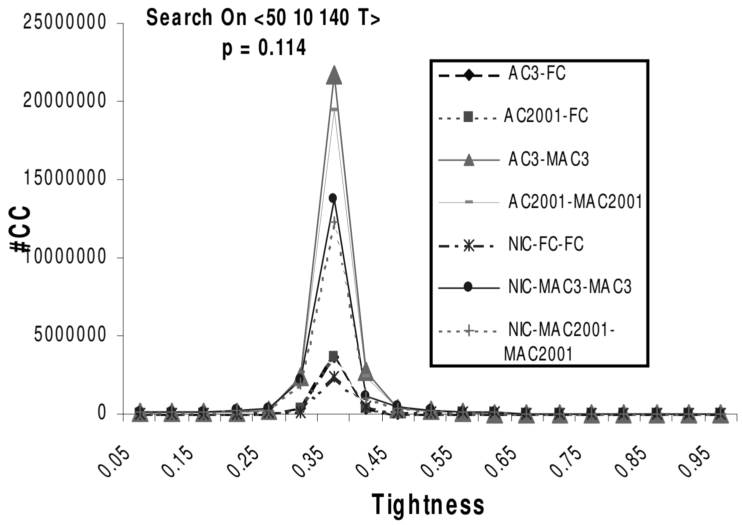


Figure 5.6: *Number of constraint checks for hybrid search for p=0.114.*

AC2001-FC consumed CPU time are 134,193ms and 130,251ms, respectively, the difference is 3942ms (3%). It is insignificant.

5.1.3 Varying constraint probability

Figures 5.9 and 5.10 show that, when varying constraint probability with fixed constraint tightness, if the constraint probability is higher than 0.2, costs of NIC based hybrid search are gradually increased, but not AC based hybrid search algorithms.

When the constraint tightness is relatively low 0.3 (see Figure 5.9), in the areas of the phase transitions of the hybrid search algorithms, NIC-MAC3-MAC3 and NIC-MAC2001-MAC2001 perform worst, AC3-FC and AC2001-FC performance best, NIC-FC-FC follows. However, when the constraint tightness is relative high (t=0.5), NIC-FC-FC has the best performance, AC3-MAC3 and AC2001-MAC2001 perform the worst. MAC based hybrid search algorithms in both conditions perform poorly. However, when the constraint tightness is high (0.7), see Figure 5.11, when the constraint probability is between 0.02 and 0.15, NIC has the worst performance, AC3-FC and AC2001-FC follows. We can conclude

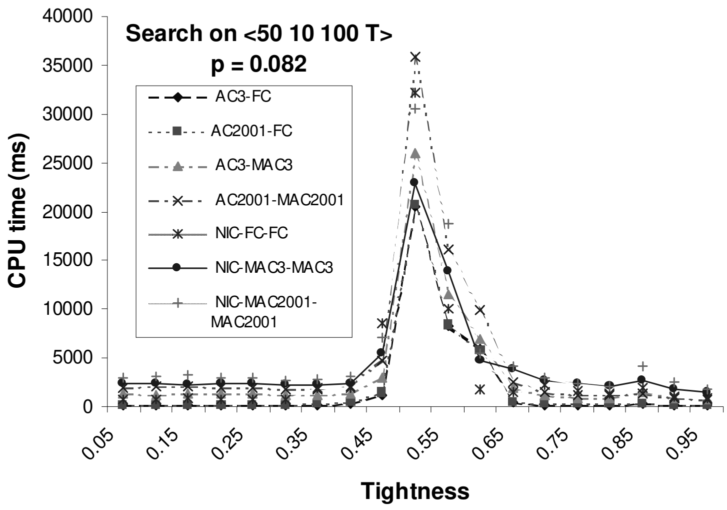


Figure 5.7: *CPU time for hybrid search process (when p=0.082).*

that, when constraint tightness is high and constraint probability is low, MAC based hybrid search algorithms perform better than FC based hybrid search algorithms. The result of experiments are given in Table 5.5.

5.1.4 Comparison of the number of nodes visited

Node visited records the number of backtracks during search process. When using node visited as measurement to compare the cost of the hybrid search algorithms, there are three concerns:

1. We do not count node visited by NIC.

2. Preprocessing processes only reduce the search space of CSP, then feed it to each search algorithm.

3. Since there are no difference of node visited by search algorithms MAC-AC3 and MAC-AC2001, we use MAC-AC3 as representative.

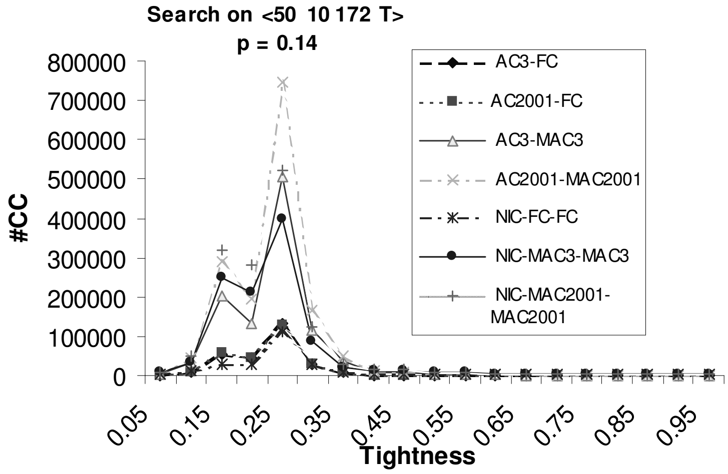


Figure 5.8: *CPU time [ms] hybrid search for p=0.14.*

From Figure 5.12, Figure 5.13 and Figure 5.14, we observed that MAC based hybrid search algorithms always have fewer node visited then the equivalent FC based search algorithms, such as AC3-MAC3 has fewer node visited than AC3-FC, NIC-MAC3-MAC3 has fewer node visited than NIC-FC-FC. The comparison between AC3-FC and AC3-MAC3 is given in Table 5.6, in this table, we can see AC3-MAC3 always has less node visited than AC3-FC, the difference is significant. And Similarly, NIC-MAC3-MAC3 has always fewer node visited than NIC-FC-FC. The comparison of node visited between AC3-FC and NIC-FC-FC is given in Table 5.7, from data, it is clear that NIC-FC-FC has less node visited than AC3-FC, except when p = 0.1, NIC-FC-FC has 7 folds node visited than AC3-FC does. Since NIC reduced search space in large amount of degree.

We also observed that NIC based hybrid search algorithms have fewer node visited than the equivalent AC based hybrid search algorithms, such as NIC-FC-FC has fewer node visited than AC3-FC, similarly, NIC-MAC3-MAC3 has fewer node visited than AC3-MAC3. Since NIC can reduced more search space than AC does, except when constraint probability is low, such as 0.1, NIC-FC-FC has the most node visited.

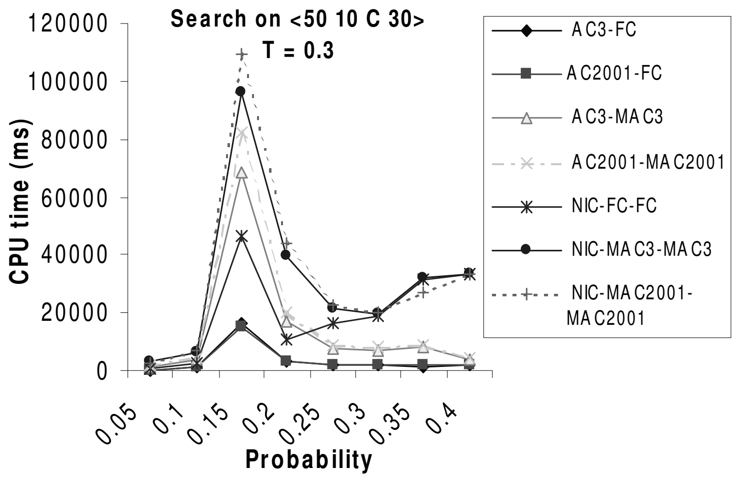


Figure 5.9: *CPU time [ms] hybrid search for T=0.3.*

5.2 Discussion

Going into this study, we excepted that, when constraint probability is low, the hybrid search algorithms combined with NIC during preprocessing would have better performance than the equivalent AC hybrid algorithms. We expected this because when the constraint probability is low, the constraint graph is sparse (the degree of the variables is low), and NIC does not need to do much work in checking consistency. Furthermore, NIC can reduce the search space to a higher degree (even when the CSP is sparse) than AC algorithms, resulting in better performance on search. However, as constraint probability increases, the number of neighbors a variable also increases, increasing the work done by NIC during consistency checks. In the extreme case, when the constraint graph is complete, the procedure of consistency checks of NIC is equivalent to a search for all solutions. Thus NIC based hybrid search algorithms take more CPU time and constraint checks than AC based hybrid search algorithms.

We now discuss why a FC-based search algorithms outperforms an equivalent MAC-based hybrid search algorithms. Although MAC generates smaller search trees than does

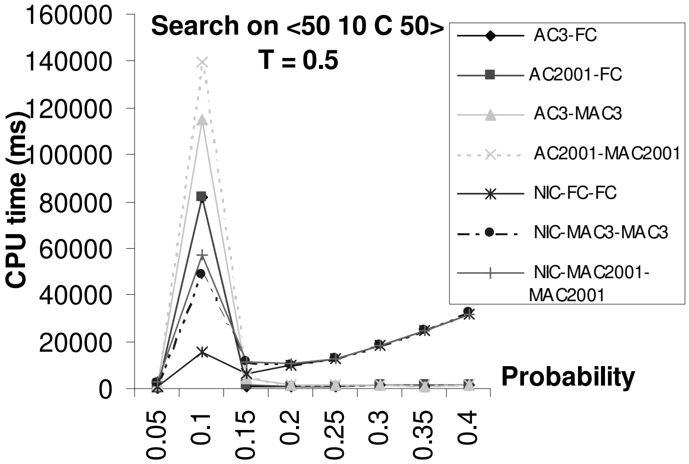


Figure 5.10: *CPU time after [ms] hybrid search for T=0.5.*

FC, for each node, MAC does full propagation among future variables, which offsets the advantage of expanding small tree. The higher the constraint probability, the higher the cost of MAC. When the constraint probability is high, every time a value is assigned to a variable, the constraint check count will be much higher than FC, which uses only partial propagation. In our study, If FC outperforms MAC, not only in the single hybrid equivalent algorithm, but for all equivalent hybrid algorithms, at the condition of constraint probability is higher than 0.1.

However, when both constraint probability is low, and the constraint tightness is high, then propagation is cheap, and MAC based hybrid algorithms perform better than FC based. The results of our experiments show that FC based search algorithms perform much better than MAC based hybrid algorithms except for the case when constraint probability is lower than 0.1, and constraint tightness larger than 0.7, NIC-FC-FC has the worst preformance.

Our study elicitates the conditions under which hybrid MAC algorithms and hybrid FC algorithms outperform each others, thus clarifying the results obtained by Sabin and Freuder [?] who argued that MAC is superior to FC. Their conclusion is correct under the

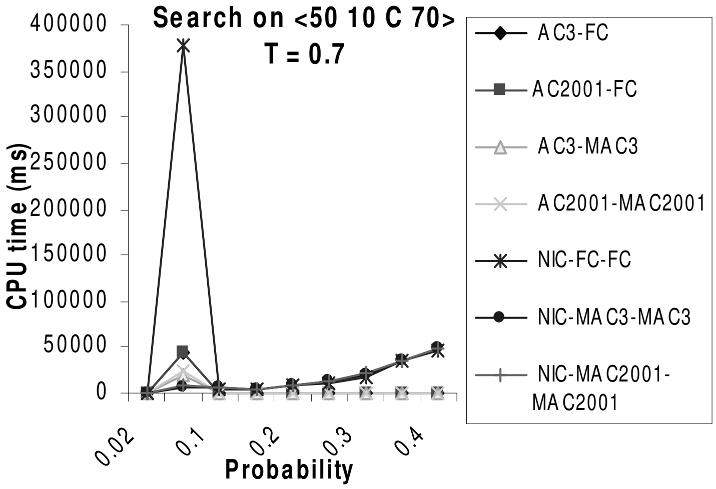


Figure 5.11: *CPU time [ms] hybrid search for T=0.7.*

specified conditions of low constraint density and high constraint tightness.

Actually the hybrid search algorithms are divided into three stages to solve CSPs (Figure 5.15, left). Figure 5.3 shows that, when the constraint tightness is low, the number of constraint checks is high (for detail see Table 5.2, when p = 0.28). The reason for this is that in the first stage, when constraint tightness is low (less than 0.5), preprocessing algorithms are ineffective in reducing the search space. The search algorithms do most of the work in solving the CSPs. In this stage, preprocessing is unnecessary. In the second stage (see Figure 5.15, right) when 0.5$< t <$ 0.7, both preprocessing and search algorithms do about the same amount of the work. In this stage, choosing hybrid lookahead search algorithms to solve CSPs is the best choice. In the third stage, when the constraint tightness is higher than 0.7, preprocessing algorithms do almost all of the work to solve the problems, and the search algorithms do little work. In this stage, choosing hybrid lookahead search algorithms to solve CSPs, however, with low level of preprocessing algorithms.

Since NIC reduces larger amount of search space than AC does, generally NIC-based search have fewer node visited than AC-based hybrid search algorithms.

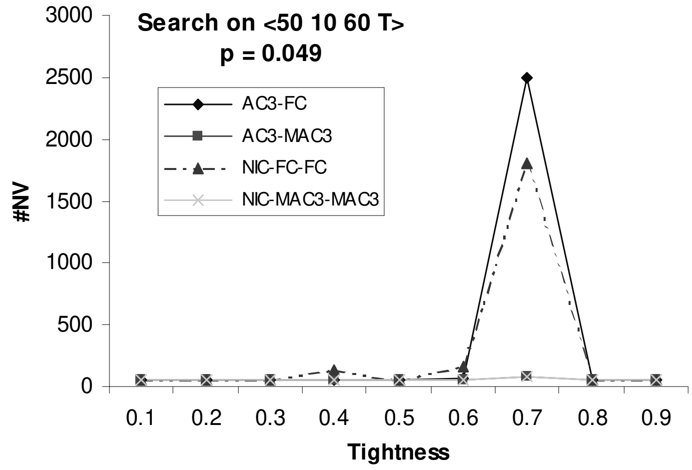


Figure 5.12: *Node visited hybrid search for p=0.049.*

MAC based search algorithms always have fewer node visited than FC based hybrid search algorithms, since for each instantiation, MAC does AC checks among the future variables, it can further reduce the future variables' domains.

We conclude that the preprocessin algorithms do not necessary when the constraint tightness is lower than 0.05 (see the Figure 4.13).

5.3 Conclusions

The results of our experiments show that FC-based hybrid search algorithms outperform MAC-based hybrid search algorithms except in the region of when the constraint tightness is high and constraint probability is low.

In all other cases, MAC is time consuming. We cannot afford the time consumed by MAC.

MAC based search algorithms always have fewer node visited than FC based hybrid search algorithms.

When constraint probability $p<0.05$, although MAC-based hybrid search algorithms

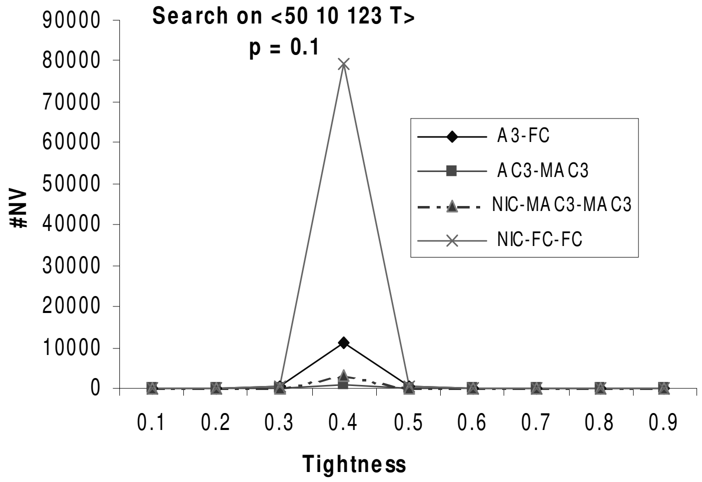


Figure 5.13: *Node visited hybrid search for p=0.1.*

cost relative low than equivalent FC-based hybrid search algorithms. However, in this region, neigher NIC or AC preprocessing can reduce search space significant, except when constraint tightness is high, such as $t > 0.7$ (see Figure 4.13), thus preprocessing is unnecessary.

When constraint probability is between (0.05< p <0.15), NIC-based hybrid algorithm should be chosen as the search algorithms. When the constraint tightness is too low, e.g. lower than 0.5, preprocessing algorithms are ineffective. In this region, preprocessing is unnecessary.

When $p > 0.15$, AC3-FC should be choosen as lookahead hybrid search algorithm to solve CSPs.

When p> 0.2, NIC-based hybrid algorithms are too costly to be used to solve CSPs.

Using AC3-based hybrid algorithms always spend less CPU time than AC2001. And AC2001 as preprocessing based hybrid algorithms always had fewer number of constraint checks than AC3 based preprocessing.

The choice for choosing look-ahead search algorithms is shown in Table 5.9.

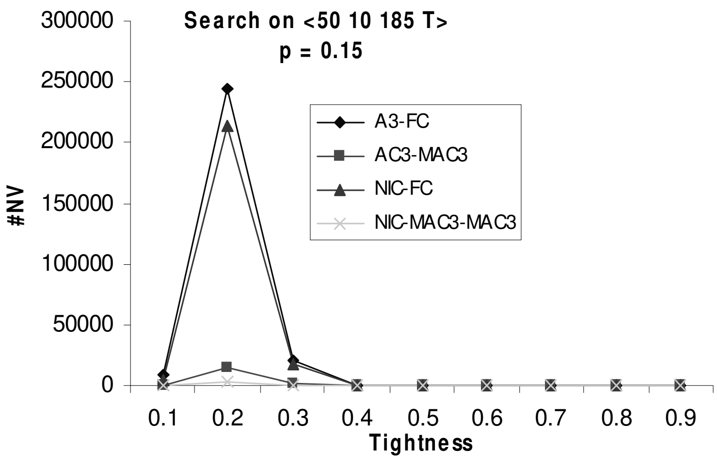


Figure 5.14: *Node visited hybrid search for p=0.15.*

Note that, X is either 3 or 2001, and $t > 0.05$. Since when $0.05 \leq p < 0.1$, there are no significant difference among NIC-MAC-ACx-MAC-ACx and ACx-FC, both of them are scaled good.

5.4 Summary

In this chapter, we tested the performance of seven hybrid search algorithms based on AC3, AC2001 and NIC. We gave the tables and figures to show the results of the experiments. The results of the experiments supported our expectations that FC-based hybrid algorithms have better performance than MAC-based hybrid algorithms except on sparse and high constraint tightness of CSPs. This study sheds a new light on the generalizations of "MAC is better than FC" of Sabin and Freuder [**?**; **?**].

We also show that NIC based hybrid algorithms are more favorable than either AC3 or AC2001 based hybrid algorithms when constraint probability is low in between [0.05, 0.1], but not smaller than 0.05.

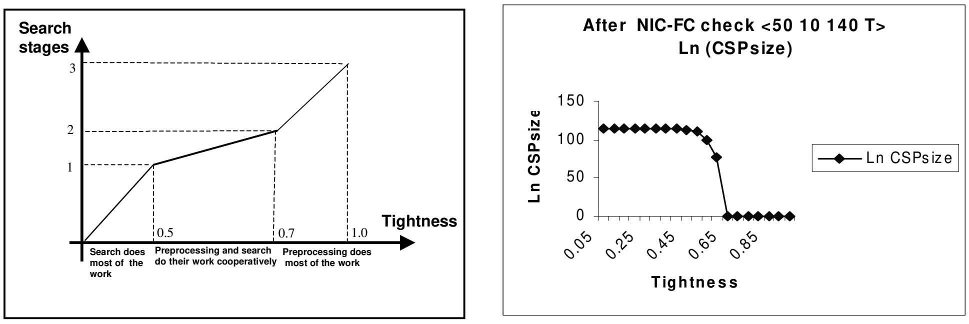


Figure 5.15: *Three stages of hybrid algorithms (left). When varying constraint tightness, size of the CSP changes (right).*

Number of constraint checks							
p = **0.114**							
	AC3		AC2001		NIC		
Constraint tightness	FC	MAC	FC	MAC	FC	MAC-AC3	MAC-AC2001
0.05	5618	58842	5592	56408	34479	132450	129258
0.1	6203	64916	6203	61327	35392	145061	140389
0.15	14785	90866	14771	84116	41135	169648	161052
0.2	23565	167028	23526	153715	46824	239468	223828
0.25	42083	227307	41991	206252	114580	304325	280961
0.3	320000	2335797	319935	2103706	113281	2120618	1916689
0.35	3594013	2.17E+07	3593727	1.94E+07	2378922	1.38E+07	1.23E+07
0.4	307157	2691883	306758	2397543	439346	1113355	999784
0.45	51197	365357	50519	325141	76374	440329	396307
0.5	28090	194799	26964	170034	58270	212036	196845
0.55	14242	75857	12343	65260	57986	122937	116647
0.6	12569	51994	9468	43586	60260	67624	65108
0.65	13188	31778	9109	25347	37840	38681	37743
0.7	10239	18371	8335	15481	34942	33734	33025
0.75	7143	12629	6525	11700	21962	25553	25224
0.8	5104	10232	4788	9680	16775	22136	21920
0.85	3890	8818	3701	8540	14258	20436	20277
0.9	2862	7499	2736	7361	12359	19752	19599
0.95	2515	7068	2404	6956	11705	19040	18929
p = **0.28**							
0.05	140624	1611257	140524	1508137	282991	2976962	2770806
0.1	65529	650708	65465	606587	281483	2301885	2107818
0.15	35113	339465	34978	314853	270976	1601832	1460083
0.2	32960	251349	32751	233809	186129	731200	662964
0.25	24088	155958	23887	142837	119114	460694	417200
0.3	21886	127270	21464	117339	86509	305890	279046
0.35	21224	100666	20014	90475	75415	220062	203417
0.4	20920	70544	18841	63264	58940	180603	168183
0.45	22135	59110	18343	51920	52332	157247	147622
0.5	25156	48922	18084	39398	47225	129536	123034
0.55	26946	49826	19336	39499	49252	130161	134268
0.6	17749	29824	16000	27391	38967	113941	108956
0.65	12903	24388	12110	23322	37614	109809	105279
0.7	8976	19894	8544	19424	33656	102540	98611
0.75	6158	17111	5912	16851	30278	98975	95397
0.8	4555	15349	4372	15164	29010	95855	92436
0.85	3608	14524	3468	14384	27124	92132	89155
0.9	3477	14325	3338	14185	25779	91422	88786
0.95	3369	14229	3235	14095	25978	87794	85625

Table 5.2: *Number of constraint checks for hybrid search for p = 0.114, and 0.28.*

Comparing of number of constraint checks of AC3-FC and NIC-FC-FC					
Lookahead algorithms	p = 0.049 t = 0.7	p = 0.65 t = 0.4	p = 0.114 t = 0.35	p =0.15 t = 0.2	p = 0.28 t = 0.05
AC3-FC	318190	974817	3594013	2123527	140624
NIC-FC-FC	87214	328120	2378922	4521114	282991
Difference	230976	646697	1215091	-2397587	-142367
Ratio	3.65	2.97	1.51	0.47	0.5
Comparing of number of constraint checks of AC3-FC and AC3-MAC3					
AC3-FC	318190	974817	3594013	2123527	140624
AC3-MAC3	118717	3454415	21741682	16634106	1611257
Difference	199473	-2479598	-18147669	-14510579	-1470633
Ratio	2.68	0.28	0.16	0.13	0.09
Comparing of number of constraint checks of NIC-FC-FC and NIC-MAC3-MAC3					
NIC-FC-FC	87214	328120	2378922	4521114	282991
NIC-MAC3-MAC3	82451	750026	13757758	16944260	2976962
Difference	4763	-421906	-10778836	-12423146	-2693971
Ratio	1.06	0.44	0.17	0.27	0.1
Comparing of number of constraint checks of AC3-FC and AC2001-FC					
AC3-FC	318190	974817	3594013	2123527	140624
AC2001-FC	316470	974430	3593727	2123422	140524
Difference percentage	1720 0.5%	387 0.04%	286 0.008%	105 0.005%	100 0.07%

Table 5.3: *Comparing of number of constraint checks at the peak of phase transitions.*

CPU time (ms)							
p = **0.082**							
	AC3		AC2001		NIC		
Constraint tightness	FC	MAC	FC	MAC	FC	MAC-AC3	MAC-AC2001
0.05	165	1363	204	1989	807	2345	3021
0.1	126	1225	196	2112	806	2319	3074
0.15	136	1308	197	2140	877	2274	3202
0.2	158	1354	207	1979	811	2356	2932
0.25	143	1352	175	1857	697	2297	2936
0.3	122	1232	187	1805	674	2172	2712
0.35	159	1179	234	1778	1989	2202	2824
0.4	313	1367	391	2006	1896	2421	3132
0.45	1193	2954	1406	4757	8590	5450	7134
0.5	20488	25958	20669	35821	32235	22944	30488
0.55	8334	11514	8385	16045	10096	13947	18755
0.6	5837	6979	5716	9929	1813	4738	5472
0.65	385	1867	475	2522	1510	3768	4173
0.7	207	1224	279	1547	2220	2718	2983
0.75	208	943	275	1193	1611	2300	2374
0.8	183	856	226	1166	1310	2002	2080
0.85	295	1522	287	1339	1851	2657	4127
0.9	192	902	158	924	1060	1735	2521
0.95	116	702	137	797	890	1542	1703
p = **0.14**							
0.05	488	2962	615	4233	2168	6708	8782
0.1	8943	31983	8305	44143	11870	33796	47993
0.15	55552	202368	56032	289989	28430	250207	319364
0.2	46336	130943	45708	193630	29436	212943	280467
0.25	134193	504011	130251	745452	113569	396132	523100
0.3	27502	115327	27551	164207	29527	87282	122719
0.35	7054	36352	7146	50976	9220	21028	28505
0.4	1794	9187	1907	12963	5086	13809	17313
0.45	1075	6687	1199	9362	3526	13130	15491
0.5	441	3144	566	4042	4242	9018	10018
0.55	384	2652	579	3494	5236	6507	7213
0.6	358	2163	521	2633	4316	5592	6069
0.65	394	1827	515	2101	3747	5245	5459
0.7	363	1547	428	1750	3301	4592	4903
0.75	274	1367	423	1662	3017	4218	4509
0.8	252	1298	270	1463	2834	4054	4327
0.85	221	1288	258	1387	2682	4005	4189
0.9	200	1187	242	1345	2528	3801	3988
0.95	214	1197	192	1318	2578	3674	3937

Table 5.4: *CPU time [ms] for hybrid search for p = 0.082, 0.14.*

CPU time (ms)							
t = **0.1**							
	AC3		AC2001		NIC		
Constraint probability	FC	MAC	FC	MAC	FC	MAC-AC3	MAC-AC2001
0.05	250	1675	215	1980	660	2085	2500
0.1	325	2690	355	3215	1700	4975	5955
0.15	151100	649905	204220	668715	251530	704720	791420
0.2	282920	1826160	283200	2076840	287290	1831710	2096890
0.25	62475	420290	62700	484165	76235	161150	185950
0.3	11260	85930	11425	100795	41545	54490	60470
0.35	6650	46165	6865	53445	38555	45780	47625
0.4	3680	21910	3900	25050	49950	59455	61055
p = **0.3**							
0.05	190	1405	220	1730	660	2880	2310
0.1	1070	3790	1155	4505	2495	6095	6650
0.15	16225	68245	15160	82270	46580	96035	109055
0.2	2990	17240	3050	20270	10930	39415	43995
0.25	1815	7530	2030	8870	16090	21370	22850
0.3	1615	6625	1725	7885	18810	19665	20020
0.35	1540	8025	2060	9090	31225	31770	27110
0.4	1595	3770	1700	4225	32985	33530	33370
p = **0.5**							
0.05	165	1430	245	1870	550	1945	2745
0.1	81510	115100	81615	139590	15820	48470	56795
0.15	960	4360	1070	4860	6505	11370	11370
0.2	795	1315	850	1620	9695	9725	10490
0.25	935	1135	960	1175	12880	12630	12580
0.3	1065	1150	1105	1210	18120	18020	18100
0.35	1015	1015	1150	1150	24525	24360	24445
0.4	1290	1265	1460	1485	31970	32400	31970
p = **0.7**							
0.01	150	180	150	185	980	760	790
0.05	43120	20760	43200	24690	377750	7030	8075
0.1	280	245	295	265	4695	5550	5610
0.15	195	205	195	210	3980	4615	4610
0.2	185	195	190	220	8560	8770	8800
0.25	140	180	155	190	11860	12280	12460
0.3	160	170	165	170	18495	19840	22880
0.35	160	190	195	195	34600	36140	36155
0.4	165	180	175	195	45670	47630	47875

Table 5.5: *CPU time [ms] for hybrid search for t = 0.1, 0.3, 0.5, 0.7.*

Comparing of the number of NV of AC3-FC and AC3-MAC-AC3			
Lookahead algorithms	$p = 0.049$ $t = 0.7$	$p = 0.1$ $t = 0.4$	p 0.15 $t = 0.2$
AC3-FC	2504	11217	244547
AC3-MAC-AC3	78	737	14044
Difference	2426	10480	10413
Ratio	32	15	17
Comparing of the number of NV of NIC-FC-FC and NIC-MAC-AC3			
NIC-FC-FC	1807	79239	213333
NIC-MAC-AC3	78	3105	3577
Difference	1729	76134	209756
Ratio	23	25	59

Table 5.6: *Comparing of the number of NV of MAC-based and FC-based techniques.*

Comparing of the number of NV of AC3-FC and NIC-FC-FC			
Lookahead algorithms	$p = 0.049$ $t = 0.7$	$p = 0.1$ $t = 0.4$	p 0.15 $t = 0.2$
AC3-FC	2504	11217	244547
NIC-FC-FC	1807	79239	213333
Difference	697	-68022	31214
Ratio	1.39	0.14	1.15

Table 5.7: *Comparing of the number of NV of* `AC3-FC` *and* `AC3-MAC-AC3`.

Number of node visited				
p = **0.049**				
constraint tightness	AC3-FC	AC3-MAC3	NIC-FC-FC	NIC-MAC3-MAC3
0.1	50	50	50	50
0.2	50	50	50	50
0.3	50	50	50	50
0.4	50	50	130	50
0.5	50	50	50	50
0.6	66	53	159	50
0.7	2504	78	1807	78
0.8	56	56	58	55
0.9	51	50	50	50
p = **0.1**				
0.1	62	50	62	50
0.2	73	51	73	51
0.3	325	174	666	173
0.4	11217	737	79239	3105
0.5	546	57	664	54
0.6	93	51	136	50
0.7	51	50	50	50
0.8	51	50	50	50
0.9	50	50	50	50
p = **0.15**				
0.1	8605	588	4556	588
0.2	244547	14044	213333	3577
0.3	21196	1334	18064	213
0.4	573	87	575	62
0.5	67	55	167	50
0.6	50	50	64	50
0.7	50	50	51	50
0.8	50	50	50	50
0.9	50	50	50	50

Table 5.8: *Number of node visited for hybrid search for p=0.049, 0.1, 0.15.*

The choice for choosing lookahead search algorithms					
Look-ahead algorithms	$p < 0.05$ preprocessing unnecessary	0.05$\leq p <$0.1	0.1$\leq p <$0.15	0.15$\leq p <$0.2	$p \geq$0.2
ACx-FC	worst	good	good	best	best
ACx-MAC-ACx	bad	worst	worst	worst	good
NIC-FC-FC	good	best	best	good	bad
NIC-MAC-ACx-MAC-ACx	best	good	bad	bad	worst

Table 5.9: *The choice for choosing hybrid look-ahead search algorithm.*

Chapter 6

Summarizing our study and reflecting upon future work

6.1 The importance of this study

In this thesis, we compared preprocessing algorithms and look-ahead search algorithms by empirical evaluation. For each algorithm, we measured CPU time, counted the number of constraint checks and node visited. Before this study, no other AI people had made this relative evaluation of these algorithms with so namely detail. From this study, we bring the question of what degree of constraint propagation should be applied to lookahead algorithms, which based on constraint probability and tightness, and also answer this question by the empirical evaluation. Our results also contradict some widely believed claims in the community.

This research compared preprocessing algorithms and lookahead search algorithms. We can make recommendations of when and which algorithm would be the best choice for solving CSPs based on particular conditions. Thus when solving a CSP, people do not have to blindly choose a combination, which may yield useful of resources.

We created a library in Java (see Appendix B) of preprocessing and search algorithms. We can easily enlarge this library as further studies may require testing additional algorithms.

6.2 Conclusions

AC3 is a simple and good AC algorithm to enforce arc-consistency in CSPs. The much more complex preprocessing algorithm AC2001 is unnecessary. Unless constraint checking is costly.

NIC has better performance than both AC3 and AC2001-based preprocessing algorithms in reducing the search space. When the constraint probability is low ($0.05 < P < 0.1$), we choose NIC as the preprocessing algorithm. When the constraint probability is higher than 0.1, the AC-based algorithm is the better choice. When constraint probability is lower than 0.05, any form of preprocessing is ineffective.

FC is a good lookahead search algorithm, except in the case of low constraint probability and high constraint tightness.

6.3 Directions for future research

There are many preprocessing algorithms that were not included in this study: AC4 [**?**], and PC-1 [**?**], PC-3 [**?**; **?**], etc. As we mentioned in Chapter 2, systematic search has two schemes, look-ahead and backtrack. In this study, we only studied look-ahead search scheme. We did not study backtrack search scheme, such as backjumping (BJ) and conflict-backing-jumping (CBJ) algorithms [**?**].

The additional study would be to compare more preprocessing algorithms, such as AC4, AC6, PC-1, PC-3, and combine them with backtrack search algorithms to find good combinations.

It would also be of interest to try to generate larger CSPs, such as those having more than 100 variables, and to study non-binary CSPs, those which contain constraints apply to more than two variables.

The present study is based on randomly generated, uniform instances and the topology of constraint networks does not included.

As we mentioned in chapter 4, AC2001 might be more efficient than AC3 when the

CSP is extremely sparse with extremely large domain sizes. To generate CSP instances with large domain size would be an open-ended.

6.4 Closing remarks

In this study, we tested five preprocessing algorithms and seven hybrid search algorithms. The results show us that CSPs can be solved more efficiently than simple exhaustive search. To combine preprocessing and search can significantly save CPU time and number of constraint checks. The question is how should we combine them and which combination is the best under certain conditions (such as a given constraint tightness and constraint probability). If they are combined in the proper way, the resulting hybrid algorithm will be more powerful. If they are combined in the wrong way, the resulting hybrid algorithm may be worse than the simple search algorithms. Optimal combination of preprocessing and look-ahead techniques is different for different problems, and testing and finding good hybrid search algorithms for solving CSPs is an interesting and useful topic of investigation.

Appendix A

Java package

To conduct our study on preprocessing and search techniques, we have built a Java package that includes 21 Java classes. We discuss this Java package in this appendix.

A.1 Data structures for the CSP

In the Java library, all information relating to a CSP is contained in the data structure CSP which is in file CSP.java.

Firstly, we use random CSP generator to create CSPs. For each CSP consist of Variable.java which contains Domain.java, and constraint.java which contains Pair.java to remember each variable-value pair. (see Figure A.1

A.1.1 Main methods in Java classes

1. public class Generator

   It extends java.lang.Object

   Its constructor needs four input parameters (see Chapter 3),

   `public Generator(int n, int d, int c, int t)`

   Parameters:

   int - N number of variables

   int - D max domain size

   int - C number of constraint

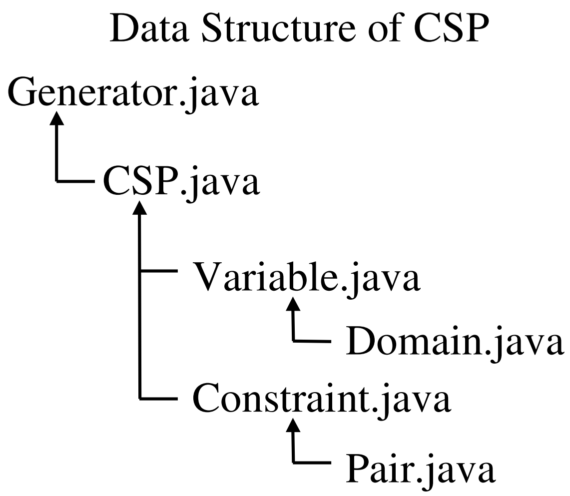


Figure A.1: *Data structure of CSP*

int - T number of allowed tuples

It includes the following methods:

- `public void setSolution()`

  To generate a solution, which can be inserted in CSP to guarantee at least one solution solution is randomly generated.

- `public void setConstraints()`

  To set a set of constraints randomly

  To ensure $i < j$ in C_{ij} and no C_{ji} in the same set

- `public void adjustConstraints()`

  To adjust constraint set to insert a solution

- `public void show()`

To show generated CSP on standard output (Monitor)

number of variables

variable + domain content

i,j (in constraint C_{ij}) + constraint tuples

- `public void showSolution()`

  To show solution on standard output (Monitor)

- `public zyang.csp.project.CSP cspInstance()`

  To produce an instance at a given N D C T condition, which is also saved in file

- `public zyang.csp.project.CSP cspInstance(java.lang.String filename)`

  To produce an instance at a given N D C T condition, which is also saved in file

2. public class CSP

   It extends java.lang.Object

   constructors:

   `public CSP(java.lang.String filename)`

   parameter:

   filename - a CSP instance

   `public CSP (java.util.ArrayList vlist, java.util.ArrayList clist)`

   parameters:

   vlist - list of variables

   clist - list of constraints

   It includes the following public methods:

   - `public java.lang.Object clone()`

     To duplicate the CSP object

     Overrides:

clone in class java.lang.Object

Returns:

CSP object

- `public void showVarList()`

  To print out the variable list on the standard output

- `public void showConstrList()`

  To print out the constraint list on the standard output

- `public java.util.ArrayList getVarList()`

  To get the variable list

  Returns:

  the variable list.

- `public java.util.ArrayList getConstrList()`

  To get the constraint list

  Returns:

  the constraint list.

- `public java.util.ArrayList getNeighbors`
  `(zyang.csp.project.Variable var)`

  To return the neighboring Variables

  Parameters:

  var - a variable.

  Returns:

  a list of variables.

- `public java.util.ArrayList subConstrList`
  `(zyang.csp.project.Variable var)`

  To get a list of constraints among the neighboring variables

  Parameters:

var - a variable.

Returns:

a list of constraints.

- `public int getDegree(zyang.csp.project.Variable var)`

  To get the degree of a variable

  Parameters:

  var - a variable.

  Returns:

  the degree.

- `public int[] CSPsizeArray()`

  To get the size of csp

  Returns:

  an array of the domain size of each variable

- `public java.lang.String CSPsizeString()`

3. `public class Constraint`

   It extends java.lang.Object

   constructor:

   `public Constraint(int i, int j)`

   Parameters:

   i - the index of the first variable.

   j - the index of the second variable.

   It includes the following public methods:

   - `public void copy(Constraint constr)`

     To duplicate a constraint object

     Parameters:

     constr - clone the constraint.

- `public int getI()`

  To get the index of the first variable

  Returns:

  the index of the first variable.

- `public int getJ()`

  To get the index of the second variable

  Returns:

  the index of the second variable.

- `public void add(zyang.csp.project.Pair p)`

  To add a new tuple in the constraint

  Parameters:

  p - tuple of constraint.

- `public int size()`

  To get the number of tuples in the constraint

  Returns:

  the number of tuples.

- `public zyang.csp.project.Pair get(int i)`

  To get the i^{th} tuple in the constraint

  Parameters:

  i - the i^{th} tuple.

  Returns:

  the tuple

- `public boolean support(java.lang.Integer obj1, java.lang.Integer obj2)`

  To check if two values are supported by this constraint or not

  Parameters:

  obj1 - value one

  obj2 - value two

Returns:

boolean

4. public class Domain

   It extends java.lang.Object

   Constructor:

   `public Domain()`

   It includes the following public methods:

   - `public void add(int i)`

     To add new value to domain

     Parameters:

     i - int value

   - `public void add(java.lang.Integer I)`

     To add new value to domain

     Parameters:

     i - Integer value

   - `public java.lang.Integer get(int i)`

     To get a value in the domain

     Parameters:

     i - index

     Returns:

     an Integer value

   - `public java.util.ArrayList copy()`

     To duplicate a list of values in the domain

     Returns:

     a list of values

   - `public int size()`

To get a domain size

Returns:

size of the domain

5. public class Pair

   It extends java.lang.Object

   Constructor:

   `public Pair(int p1, int p2)`

   Parameters: p1 - The index of the first variable of the constraint tuple.

   p2 - The index of the second variable of the constraint tuple.

   It includes the following public methods:

   - `public java.lang.Integer getV1()`

     To get the index of the first variable in the constraint tuple

     Returns:

     an Integer

   - `public java.lang.Integer getV2()`

     To get the index of the second variable in the constraint tuple

     Returns:

     an Integer

6. public class Queue

   It extends java.lang.Object

   Since there is no queue structure is provided by Java 2, I used two stacks to build a queue - learned from unl/cse310, Instructor Dr. Berthe Y. Choueiry. It includes the following public methods:

- `public void enQueue(java.lang.Object obj)`

  EnQueue operation

  Parameters:

  obj

- public java.lang.Object deQueue()

  DeQueue operation

  Returns:

  object

- `public java.lang.Object peek()`

  To check the tail object of the Queue

  Returns:

  object

- `public boolean empty()`

  To check the Queue is empty or not

  Returns:

  boolean value

- `public boolean contains(java.lang.Object obj)`

  To check does the Queue contain the object

  Parameters:

  obj - the tail object

  Returns:

  boolean value

A.2 Preprocessing algorithms

For studying the preprocessing algorithms, we wrote some selected representative preprocessing algorithms, such AC3, AC2001, NIC-FC-DVO, NIC-MAC-DVO-AC3, and NIC-MAC-DVO-AC2001. In the Java code library, we wrote two types of AC3 and AC2001 algorithms, AC3v denotes variable based AC3 algorithm, AC3c denotes constraint based AC3 algorithm. Similarly, AC2001v denotes variable based AC2001 algorithm, AC2001c denotes constraint based AC2001 algorithms. In this study, since we only used variable based AC algorithms, we omitted 'v' in both variable based AC3 and AC2001 algorithms.

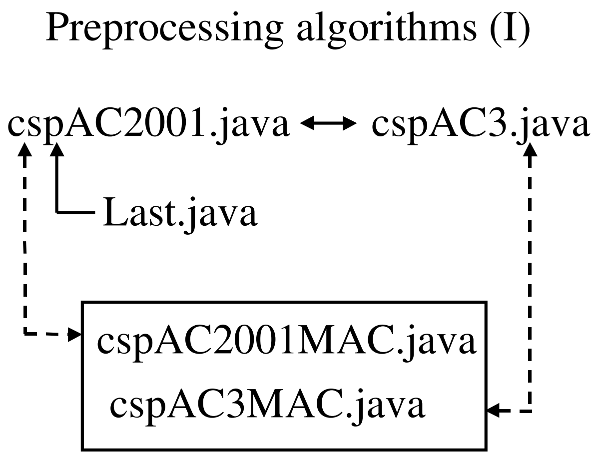


Figure A.2: *Preprocessing algorithms I*

In all figures in Appendix A, we did not make the distinction between AC3v, and AC3c, between AC2001v and AC2001c, they are all written as AC3, or AC2001. Figure A.2 shows that if the data structure `Last` is added to AC3, we have AC2001.

Since in NIC algorithm, the specified lookahead search algorithms are needed to be applied. We frame the algorithms in Figures A.2 to A.5 borrowed from search algorithms,

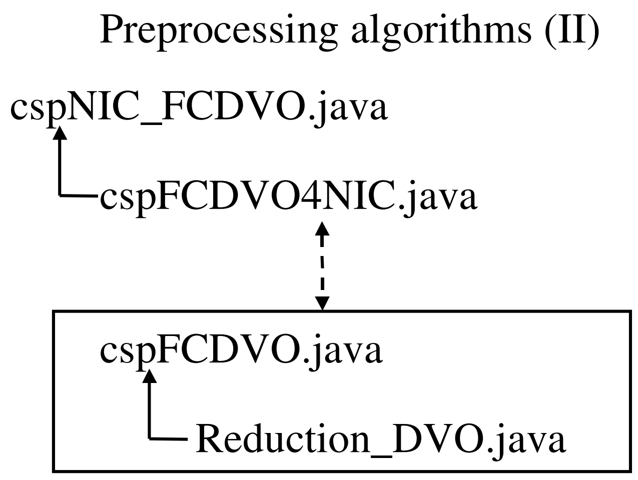


Figure A.3: *Preprocessing algorithms II*

which we will introduce in the section A.4. In Figure A.2 and Figure A.3, cspAC3MAC.java and cspAC2001MAC.java are two algorithms written specified for MAC, in which the main code is same, the only difference are the interfaces were added to cspAC3MAC.java and cspAC2001MAC.java. Similarly, in Figure A.3 and Figure A.4, cspFCDVO4NIC.java and cspMACDVOAC3NIC.java.

For instance, a specified class cspFCDVO4NIC is written for NIC with specified interface. Since in NIC, once it assigns a value from a variable, it will check the consistency which is based on find local solutions. If there is a solution, then the algorithm returns ‘true,’ otherwise, it returns ‘false.’ Solutions are not important, but either there is solution or no solution is important. This is not same as regular FC search. In FC, each time assigns a value to a variable, the partial solution is important, it can print the partial solution in every level of search tree.

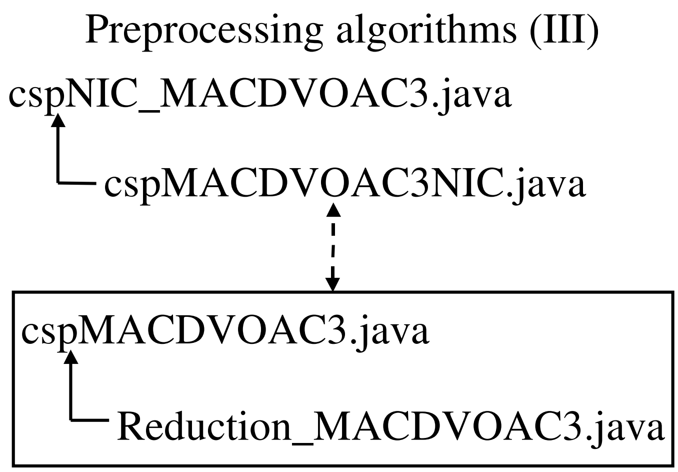


Figure A.4: *Preprocessing algorithms III*

A.3 Main methods in Java classes

1. public class cspAC3

   It extends java.lang.Object

   constructors:

   default constructor: `public cspAC3()`

   `cspAC3(zyang.csp.project.CSP input)`

   parameter:

   input - CSP instance. It includes the following public methods:

   - `public int getCHECK()`

     To get the number of constraint check

     returns:

     int - the number of constraint checks

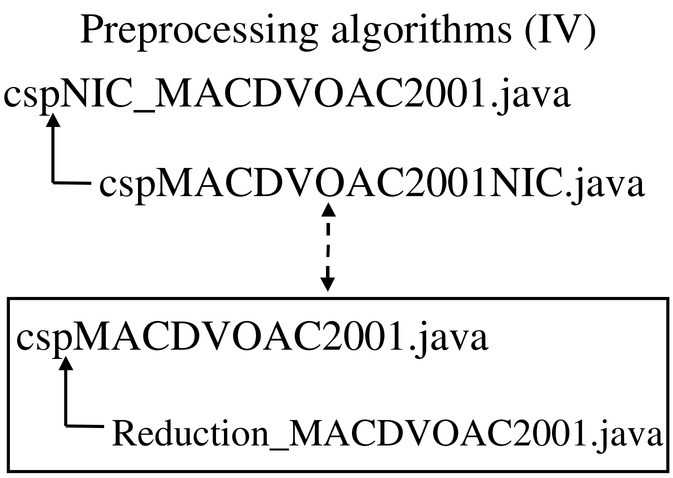


Figure A.5: *Preprocessing algorithms IV*

- `public int getNODE()`

  To get the number of node visited

  returns:

  int - the number of Node visited

- `public void showConstrList(java.util.ArrayList clist)`

  To show a list of constraints

  parameter:

  clist - list of constraints

- `public void showVar(zyang.csp.project.Variable var)`

  To show the id (identification) of the variable and its current and initial domains

  parameter:

  var - a variable

- `public void showVarList(java.util.ArrayList list)`

  To show a list of variables

  parameter:

list - a list of variables

2. `public class cspAC2001`

   It extends java.lang.Object

   Class cspAC2001 differs from class cspAC3 is that in class cspAC2001, there is an additional data structure ‘protected zyang.csp.project.Last’, see the description of class Last.

3. `public class cspNIC-FCDVO`

   It extends java.lang.Object

   constructor:

   cspNIC-FCDVO(zyang.csp.project.CSP input)

   parameter:

   input - a CSP instance

   It includes the following public methods:

   - `public void checkList(java.util.ArrayList list)`

     To print out the variable list on the standard output

     parameter:

     list - a list of variables

   - `public int getCHECK()`

     To get the number of the constraint checks

     returns:

     The number of the constraint checks

   - `public int getNODE()`

     To get the number of node visited

     returns:

     The number of node visited

   - `public java.util.ArrayList makeNeighborClone`
     `(java.util.ArrayList neighbors,`

`zyang.csp.project.Variable var,`

`java.lang.Integer val)`

To duplicate a list of neighbor variables

Parameters:

neighbors - a list of neighbor variables

var - the specified variable

val - the specified value

Returns:

a list of neighbor variables

- `public void NIC()`

  To implement Neighbor inverse consistency algorithm

4. public class cspNIC-MACDVO

   It extends java.lang.Object

   constructor:

   `cspNIC-MACDVO(zyang.csp.project.CSP input)`

   parameter:

   input - a CSP instance

   Note that MACDVO specified MACDVOAC3.

   The main methods included in class cspNIC-MACDVO are same as in class cspNIC-FCDVO, the only difference is in cspNIC-FCDVO, it uses specified FC search algorithm written for NIC, named cspFCDVO4NIC. But in class cspNIC-MACDVO, it uses cspMACDVO4NIC.

5. public class cspNIC-MACDVOAC2001

   It extends java.lang.Object

   constructor:

`cspNIC-MACDVOAC2001(zyang.csp.project.CSP input)`

parameter:

input - a CSP instance

The main methods included in class cspNIC-MACDVO are same as in class cspNIC-MACDVO, the only difference is in cspNIC-MACDVO, it uses specified MAC search algorithm written for NIC, which is cspMACDVO4NIC.

But in class cspNIC-MACDVOAC2001, it uses cspMACDVOCA2001NIC.

6. public class cspNIC-FCDVO4NIC

   It extends java.lang.Object

   It is specified FC search algorithm written for NIC. Its constructor needs two parameters as input. constructor:

   `cspFCDVO4NIC`

   `(java.util.ArrayList vList,`

   `java.util.ArrayList cList)`

   parameters:

   vlist - list of neighbor variables

   clist - list of constraints

   It includes the following main methods:

   - `protected void cutoff-DVO`

     `(java.util.ArrayList rest,`

     `zyang.csp.project.Reduction-DVO rdc)`

     To assist the function for Forward-checking, cutoff means to do domain reductoin.

   - `public java.util.Stack FC-DVO()`

To implemnet forward Checking algorithm

- `public void forward-DVO (java.util.Stack sol, java.util.ArrayList rest, java.util.Stack stack)`

  To check forward using dynamic variable ording (DVO)

- `public int getCHECK()`

  To get the number of checks

- `public int getNODE()`

  To get number of node visited

- `protected boolean hasEmptyDom-DVO (java.util.ArrayList rest)`

  To assist the function for Forward-checking to check DWO,

- `public boolean hasSolution()`

  To return the result: if there exists a solution

- `public zyang.csp.project.Variable next-DVO (java.util.ArrayList rest)`

  To implement dynamic variable ordering

- `protected void putback-DVO (java.util.ArrayList rest, zyang.csp.project.Reduction-DVO rdc)`

To assist the function for Forward-checking, to putback means to restore the future variable domains.

- `public zyang.csp.project.Variable rewind-DVO`
  `(java.util.Stack sol,`
  `java.util.ArrayList rest,`
  `java.util.Stack stack)`
  To rewind back or to do backtrack.
- `public void showVarList()`
  To print out the variable list on the standard output

7. `public class cspNIC-MACDVO4NIC`
   It extends java.lang.Object
   It is specified MAC search algorithm written for NIC. Its constructor needs two parameters as input. constructor:
   `cspMACDVO4NIC`
   `(java.util.ArrayList vList,`
   `java.util.ArrayList cList)`
   parameters:
   vlist - list of neighbor variables
   clist - list of constraints
   It includes the same methods as in class cspNIC-FCDVO4NIC, except in cspNIC-MAC4NIC uses full look-ahead instead of partial look-ahead in FC search.

8. public class cspNIC-MACDVOCAC2001NIC
   It extends java.lang.Object
   It is specified MAC search algorithm written for NIC. Its constructor needs two parameters as input.

constructor:

`cspMACDVOAC2001NIC(java.util.ArrayList vList, java.util.ArrayList cList)`

parameters:

vlist - list of neighbor variables

clist - list of constraints

It includes the same methods as in class cspNIC-MACDVO4NIC, except in cspNIC-MACAC2001NIC use AC2001 instead of AC3 inside MAC.

A.4 Search algorithms

Java programs written for specified lookahead search algorithms are showed in Figure A.6, we apply DVO to each search algorithm, such as cspFCDVO.java, cspMACDVOAC3.java, and cspMACDVOAC2001, in which include specified data-structure Reduction-DVO.java, Reduction-MACDVOAC3.java, and Reduction-MACDVOAC2001.java.

If we combine the search algorithms with preprocessing algorithms, we obtain the hybrid search algorithms which we have studied in this thesis.

A.4.1 Main methods in Java classes

1. public class cspFCDVO

   It extends java.lang.Object

   constructor:

   `public cspFCDVO(zyang.csp.project.CSP input)`

   parameter:

   input - a CSP instance

   It includes the following main methods:

Search algorithms

cspFCDVO.java
└─ Reduction_DVO.java

cspMACDVOAC3.java
└─ Reduction_MACDVOAC3.java

cspMACDVOAC2001.java
└─ Reduction_MACDVOAC2001.java

Figure A.6: *Search algorithms*

- `protected void cutoff-DVO`
  `(java.util.ArrayList rest,`
  `zyang.csp.project.Reduction-DVO rdc)`
  Function cut off (or reduction) for Forward-checking
  Parameters:
  rest - a list of the future variales
  rdc - all reduced variables

- `public java.util.Stack FC-DVO()`
  To implemnet forward Checking (FC-DVO) algorithm
  Returns:
  solution

- `protected void forward-DVO`
  `(java.util.Stack sol,`
  `java.util.ArrayList rest,`

`java.util.Stack stack)`

To check forward using least doamin (LD) of dynamic variable ording (DVO)

Parameters:

sol - partial soluion

rest - the list of the rest (or future) variables

stack - a temp stack

- `public int getCHECK()`

  To get the number of checks

  Returns:

  the number of constraint checks

- `public int getNODE()`

  To get the number of node visited

  Returns:

  the number of node visited

- `protected boolean hasEmptyDom-DVO`
  `(java.util.ArrayList rest)`

  To check is there a empty domain in the future variables' domains

  Parameters:

  rest - a list of the future variables

  Returns:

  a boolean to show if there exists an empty domain

- `public zyang.csp.project.Variable next-DVO`
  `(java.util.ArrayList rest)`

  To implement dynamic variable ordering (LD)

  Parameters:

  rest - a list of variables to be checked

Returns:

a variable

- `protected void putback-DVO`
  `(java.util.ArrayList rest,`
  `zyang.csp.project.Reduction-DVO rdc)`
  Function putback (or restore) for Forward-checking
  Parameters:
  rest - a list of the future variables
  rdc - all reduced variables

- `public void showVarList()`
  To print out the variable list on the standard output

2. public class cspMACDVO
   Inside MAC, it uses AC3 to do full propagation. It extends java.lang.Object.
   constructor:
   `public cspMACDVO(zyang.csp.project.CSP input)`
   Parameters:
   input - an instance of CSP
   The main methods included in class cspMACDCO are mostly same as in class cspFCDVO, except method,
   `public java.util.Stack MAC-DVO()` which to implemnet full lookahead algorithm instead of the method,
   `public java.util.Stack FC-DVO()`, which to implemnet forward Checking (FC-DVO) algorithm

3. public class cspMACDVOAC2001
   Inside MAC, it uses AC2001 to do full propagation.

It extends java.lang.Object.

constructor:

`public cspMACDVOAC2001(zyang.csp.project.CSP input)`

Parameters:

input - an instance of CSP

The main methods included in class cspMACDCOAC2001 are mostly same as in class cspMACDVO, except method,

`public java.util.Stack MAC-DVOAC2001()` which to implemnet full lookahead algorithm with AC2001 instead of method,

`public java.util.Stack MAC-DVO()`, which to implemnet full lookahead algorithem with AC3 as full propagation.

4. public reduction-DVO

   It extends java.lang.Object

   Reduction-DVO class is Designed for class cspFCDVO

   constructor:

   ```
   public Reduction-DVO
   (zyang.csp.project.Variable thisVar,
   java.util.ArrayList thisRest,
   java.util.ArrayList constrList)
   ```

   Parameters:

   thisVar - the current variable

   thisRest - a set of arrayList of the future variables

   constrList - set of arrayList of constraints

   It includes the following main methods:

   - `public int getCHECK()`

     To get the number of constraint check

     Returns:

a int value - the number of constraint check

- `public int getNODE()`

  To get the number of node visited

  Returns:

  a int value - the number of node visited

- `public java.util.ArrayList getReducedDom`
  `(zyang.csp.project.Variable nextVar)`

  To get the reduced domain

  Parameters:

  nextVar - next variable

  Returns:

  the reduced domain

- `protected void setReducedDomList`
  `(java.util.ArrayList constrList)`

  To set the list of the reduced domains

  Parameters:

  constrList - a list of constraints

- `public java.util.ArrayList subConstrList`
  `(java.util.ArrayList rest,`
  `java.util.ArrayList constrList)`

  To get subconstraint list

  Parameters:

  rest - a list of future variables

  constrList - a list of constraints

  Returns:

a subconstraint list connecting only future variables

5. public reduction-MACDVO

   It extends java.lang.Object

   Reduction-MACDVO class is Designed for class cspMACDVO with AC3 propagation.

   constructor:

   ```
   public Reduction-MACDVO
   (zyang.csp.project.Variable thisVar,
   java.util.ArrayList thisRest,
   java.util.ArrayList constrList)
   ```

   Parameters:

   thisVar - the current variable

   thisRest - a set of arrayList of the future variables

   constrList - set of arrayList of constraints

   The main methods it includes mainly same as in class reduction-FCDVO, except it includes a new method:

   ```
   public void checkAC(java.util.ArrayList constrList)
   ```

   To make AC3 check in MAC

   Parameters:

   constrList - an arrayList of constraint list

6. public reduction-MACDVOAC2001

   It extends java.lang.Object

   Reduction-MACDVOAC2001 class is Designed for class cspMACDVOA2001 with AC2001 propagation.

   constructor:

   ```
   public Reduction-MACDVOAC2001
   (zyang.csp.project.Variable thisVar,
   ```

```
java.util.ArrayList thisRest,
java.util.ArrayList constrList)
```

Parameters:

thisVar - the current variable

thisRest - a set of arrayList of the future variables

constrList - set of arrayList of constraints

The main methods it includes mainly same as in class reduction-MACDVO, except it in method:

```
public void checkAC(java.util.ArrayList constrList)
```

To make AC2001 check in MAC instead of AC3

Parameters:

constrList - an arrayList of constraint list

A.5 Diagram of the Java package

The diagram of the Java package is given in Figure A.7. Firstly, CSP Generator is used to generate CSPs, then the preprocessing algorithms are applied (left side) then combining whit search algorithms (right side). The results will be returned by running Final.java, in which calculates the average of instances obtained from same set of parameters. And in Final.java, each algorithm is tested as a project, such as AC3 as proj1.java, AC2001 as proj2.java, and so on. We tested twelve algorithms, The order is same as given in the Tables and Figures. Broad arrows respresent the relationship of classes derivation. Black arrows indicate the compling sequence, wave signs represent variation of classes.

A.6 Guide to the programs in the Java library

A.6.1 Introduction

The name of Java package is:

Java zyang.csp.project, which is written to solve CSPs.

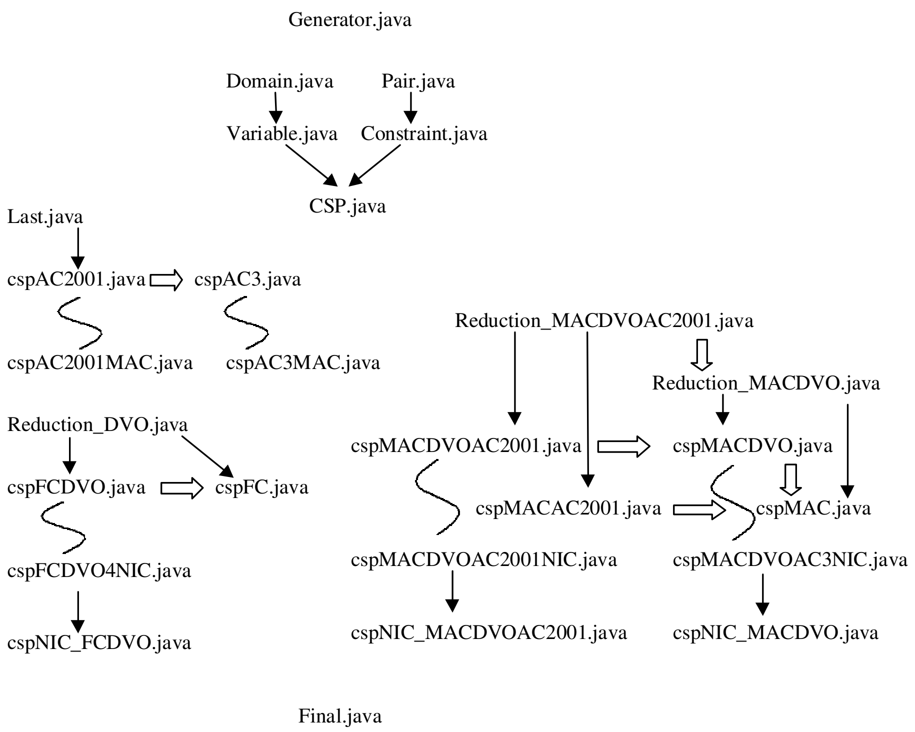


Figure A.7: *The diagram of the Java package*

Author: Zheying Yang

Data: January - April, 2003

A.6.2 To compile

1. Basic data structures:

   This is the sequence order to classes in `zyang.csp.project` package:

   - Domain.java
   - Pair.java $\rightarrow$ Constraint.java

- Variable.java
- Queue.java
- CSP.java

2. CSP Generator:

   Generator.java

3. Algorithms for arc consistency:

   - cspAC3.java
   - Last.java → cspAC2001.java

4. Search Algorithms:

   - cspFCDVO.java
   - cspAC3MAC.java → Reduction-MACDVOAC3.java → MACDVOAC3.java
   - cspAC2001MAC.java → Reduction-MACDVOAC2001.java → MACDVOAC2001.java

5. Algorithms for NIC:

   - MACDVOAC3NIC.java → cspNIC-MACDVOAC3.java
   - MACDVOAC2001NIC.java → cspNIC-MACDVOAC2001.java

6. projX.java (X=1,. . . , 15) is wrapped one, which returns the number of constrain check and CPU time.

   (a) Algorithms for constraint checking:

      - AC3 → proj1.java
      - AC2001 → proj2.java
      - NIC-FCDVO → proj3.java
      - NIC-MACDVOAC3 → proj4.java

- NIC-MACDVOAC2001 $\rightarrow$ proj5.java

(b) Search algorithms:

- FCDVO $\rightarrow$ proj6.java
- MACDVOAC3 $\rightarrow$ proj7.java
- MACDVOAC2001 $\rightarrow$ proj8.java

(c) Hybrid search algorithms:

- AC3+FCDVO $\rightarrow$ proj9.java
- AC2001+FCDVO $\rightarrow$ proj10.java
- AC3+MACDVOAC3 $\rightarrow$ proj11.java
- AC2001+MACDVOAC2001 $\rightarrow$ proj12.java
- NIC-FCDVO+FCDVO $\rightarrow$ proj13.java
- NIC-MACDVOAC3+MACDVOAC3 $\rightarrow$ proj14.java
- NIC-MACDVOAC2001+MACDVOAC2001 $\rightarrow$ proj15.java

7. Tools for calculating the averages of result values:

   Final.java

A.6.3 Running the code

1. java (option) zyang.csp.project.Generator N D C T > insYZ.dat
   will generate an CSP instance written in file (insYZ.dat).

2. java (option) zyang.csp.project.projX insYZ.dat (here X=1,. . . ,15)
   in file will show the results of number of constraint checks (`CC`) and CPU time.

3. java (option) zyang.csp.project.Final N D C T I (here I is the number of instances)
   will show the final results of all algorithms and their combinations

4. For large CSPs, add the option on, -Xmx128M -Xms64M
   will reserve more memory for Java Virtual Machine.

Appendix B

Glossaries

Arc-consistency: We say that a constraint is arc consistent (AC) if for any value of the variable in the constraint there exists a value for the other variable in such a way that the constraint is satisfied. In other words, for a given constraint graph G, arc(Vi, Vj), the edge between V_i and V_j, is arc consistent, iff for every value a in D_{V_i}, there exists a value b in D_{V_j} such that (a, b) $\in C_{V_i,V_j}$. And vice versa.

Arc consistency of a value in the domain of a variable: In a constraint C_{XY}, for a given value a in the domain of variable X should has a consistent value b in the domain of variable Y. We say the value a is supported by the value b.

Arc consistency of a variable according to a constraint: A variable V_i is arc-consistent relative to V_j iff for every value a in D_{V_i}, there exists a value b in D_{V_j} such that the constraint C_{V_i,V_j} holds.

Arc-consistency of a constraint in a CSP: A constraint $C_{V_iV_j}$ is arc-consistent, iff V_i is arc-consistent relative to V_j and V_j is arc-consistent relative to V_i.

Arc consistency of a CSP: If all constraints in a CSP are arc-consistent, then the CSP is arc-consistency.

Backtracking: In the backtracking algorithm (BT), the current variable is assigned a value from its domain. This assignment is then checked against the current partial solution;

if any of the constraints between this variable and the past variables is violated, the assignment is abandoned and another value for the current variable is chosen. If all values for the current variable have been tried, the algorithm backtracks to the previous variable and assigns it a new value. If a complete solution is found, i.e. a value has been assigned to every variable, the program may terminate if only one solution is required, or, carry on to find additional solutions. If there are no solutions, the algorithm terminates when all possibilities have been considered.

Backjumping: Backjumping (BJ) tries to avoid and reduce thrashing by jumping to the reason for failure and then proceed as in BT. That is to say, it jumps to the deepest checked against variable.

Backmarking: By keeping track of where past consistency tests failed, it tries to reduce amount of repeated consistency checking.

Bandwidth of a graph: The maximum of the pairwise minimum distance between two nodes of a graph.

Binary constraint: A constraint that involves two variables. That is, the constraint arity is equal to two.

Chronological backtracking: Is a method of solving a CSP by incrementally extending a partial solution that specifies consistent assignments for some of the variables, to a complete solution, by repeatedly choosing a value for each consective variable consistent with the values in the current partial solution. Whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of all variable domain.

Cartesian product or size of CSP: The number of possible assignments is calculated taking the product of all domain size, $\mid D_{V_1} \mid \times \mid D_{V_2} \mid \times \cdots \times \mid D_{V_n} \mid$. This is also called the size of CSP, which depends on the number of variables and size of the domains.

Chordal graph or trangulated graph: A chordal graph or triangulated graph is one in which every cycle with more than 3 edges possess a chord.

Coloring constraint: A constraint in a CSP, where,

For Set of variables X=$\{X_1, X_2, \ldots, X_n\}$, the domains for each variables are the same, D_i=$\{red, green, blue\}$;

This type of constraint states that: for each two neighbors, (X_i, X_j), where $i{\neq}j$, $X_i \neq X_j$.

Complete graph: A graph in which each pair of vertices is connected by an edge. A complete graph with n vertices is denoted K_n and has $n(n-1)/2$ undirected edges. In older literature, complete graphs are called universal graphs.

Completeness (of an algorithm): If the algorithm guarantees to find a solution if only one is needed or to find all solutions when they are exist, it is complete.

Conflict-directed-backtracking (CBJ): During the search, it jumps back to the deepest conflicting variables.

Constraint arity: The number of variables that are involved in a constraint. Ex: unary constraint has one variable, binary constraint has two variables and, ternary constraint has three variables,. . . , while global constraint contains all variables in CSP.

Constraint graph: A graph, in which each node represents a variable, and each arc represents a constraint between the variables. The node labels represent the variable domains.

Constraint network: A constraint Network is a declarative structure which consists of a number of nodes connected by constraints. A node represents an individual variable. A constraint represents a relation among the values of the nodes it connects.

Constraint scope: For a given constraint $C_{i,j,\ldots,m}$, the set of variables $\{V_i, V_j, \ldots, V_m\}$ to which the constraint applies are called the constraint'scope.

Continuous CSPs: Numerical variables with continuous domains defined over the real. Constraints that bear on numerical variables are equalities and inequalities and algebraic expressions.

Current-domain: In backtracking and hybrid backtracking algorithms, a data structure called current-domain(i) is maintained. This data structure reflects the changes in the domain of a variable after instantiation and associated constraint checks. If backtracking occurs, the current-domain(i) of future variables needs to be restored to the states they were in prior to that instantiation.

Current variable: The variable that is to be instantiated with a value from its domain. This must be consistent with the instantiations of all past variables.

Decision Problem: A problem whose answers is YES or NO.

Degree of a node in a graph: Let G be a graph and V a vertex (node) of G. The degree of V, denoted deg(V), is the number of edges that are incident with V.

Directed Acyclic Graph: A graph with:

- partially ordered set of vertices
- direct successor or descendant
- direct predecessor or ancestor
- no cycles

Directed Arc-consistency: Directed arc (V_i, V_j) is arc consistency if $\forall\ x \in D_i$, $\exists\ y \in D_j$ such that (x, y) is allowed by constraint C_{ij}.

Domain annihilation or domain wipe-out: When there is no more values left of the current assignment domain.

Finite (discrete) CSP: Discrete finite Constraint Satisfaction Problem is defined by:

- A finite set of variables V,
- A finite, set of domain D_i each containing a finite number of discrete values for each variable V_i,
- A finite set of constraints C between any pairs of variables in V .

A solution of a CSP is a consistent assignment of all variables to values in such a way that all the constraints are satisfied. A CSP can have zero solutions (insoluble), one or many solutions.

Fail first principle (FFP): Is used in variable ordering heuristic, aims at recognizing dead-ends as soon as possible, so that search effort can be saved.

Forward checking (FC): This search mechanism checks forward. It checks the consistency between the current variable and future variables which have directed constraints with the current variable. If the current value for the current variable is not consistent with future variables then this value is removed from the current-domain, and the next value in the domain is selected, until the consistency check is passed, and search proceeds the next deep level variable. If current-domain is wiped out, backtracking occurs.

Each time a variable is assigned a value, it will restrict the domains of future variables. As soon as all the variables except one of a constraint have been assigned values, all values of the remaining variable that are inconsistent with this value should be removed from their domains.

Future variable: Variables that have not yet been instantiated.

Global constraint: Constraints with maximum arity. In global consistent, any value appearing in a domain will appear in at least one solution.

Intelligent backtracking: The simplest algorithm for solving such a problem is chronological backtracking, but this method suffers from a malady known as "thrashing," in which essentially the same subproblems end up being solved repeatedly. Intelligent backtracking algorithms, such as backjumping and dependency, which improves backtracking algorithms, records the point at which each logic variable becomes bound and, when a given set of bindings leads to failure, ignores any choice point which does not bind any of those variables. No choice from such a choice point can succeed since it does not change the bindings which caused the failure.

Iterative repair: Local search technique, modifies a global but inconsistent solution to decrease the number of violated constraints.

Least domain heuristic: One of the popular variable ordering heuristic, in which the smallest domain size variable should to be assigned next.

Lookahead strategy: The methods prevent future conflicts via consistency checks among current variable and not yet instantiated variables. It includes partial look-ahead or forward checking (FC) which checks only constraints between the current variable and the direct neighbor future variables, removing the values inconsistent with the assignment of the current variable. Another strategy is full look-ahead which performs full arc consistency that will further reduces the domains and remove possible conflicts. This approach is also called maintaining arc consistency(MAC).

Maintaining arc-consistency (MAC): This technique is also called full looka-head, it performs full arc consistency that will reduce the domains and removes possible conflicts.

Map coloring problem: In a map coloring problem, the Constraints are such that no two neighbors have the same color. It is a CSP to color the vertices of a graph using a predefined number of colors so that the vertices that are connected by an edge are of different colors.

Maximum cardinality heuristic: A variable ordering heuristic, in which assignment is made according to degree of nodes. Choose a node arbitrarily. Among the remaining nodes, choose the one that is connected to the maximum number of already chosen nodes, break ties arbitrarily, repeat. Reverse the final order.

Min-conflict heuristic: A value ordering heuristics, which orders values according to the conflicts in which they are involved with the future variables.

Minimality (of a CSP): The definition of minimal CSP is concerned with binary CSPs, any consistent combination of values for 2 variables is extendable to a solution.

Node Consistency: Node i is node consistent iff for any value $x \in D_i$, predicate C$_i i(x)$ hold.

Node expansion (in search): This mechanism is in systematic search to expand on the current node.

Past variable: Variables that have already been instantiated with values from their domains and are consistent with the instantiations of all other instantiated variables.

Path consistency: A path $(V_0, V_1, \ldots, V_m)$ of length m is consistent, iff for any value x in domain of V_0 and for any value y is domain of V_m, that are consistent (i.e., $P_{V_0 V_m}(x, y)$ a sequence of values $z_1, z_2, \ldots, z_{m-1}$ in the domains of variables $V_1, V_2, \ldots, V_{m-1}$, such that all constraints among them are satisfied (i.e., $P_{V_0 V_1}(x, z_1) \wedge P_{V_1 V_2}(z_1, z_2) \wedge \ldots \wedge P_{v_{m-1} v_m}(z_{m-1}, y)$.

Pruning: FC ensures that each time a current variable is assigned a value, the domain of each future variable connected to this current variable via a constraint is revised to eliminate values which are inconsistent with the assignment of the current variable. This process is also called pruning. Because of pruning, subtrees without solution are cut off, and the drawback of *thrashing* is avoided.

Reductions(i): In FC and MAC, reductions(i) is a set of inconsistency values removed from current-domain(i) by some variable before v_i.

Resource allocation problem: A problem that deals with how many resources to allocate to a number of clients, and each of who has some number of requirement. The resources limitation restrict the allocation. Total resources should be allocated with fairness and reasonable.

Revise (procedure of the Waltz algorithm): Revise(C_{V_i,V_k,V_m,V_n}) refines the domains of V_i, V_k, V_m, V_n by iterating over these variables until quiescence.
Revise refine all the parameters $X_1, \ldots, X_k$ of a given constraint C, and returns the set of all parameters whose set was changed.

SAT: Instance: A set of literals $X = \{X_1, \neg X_1, X_2, \neg X_2, \ldots, X_n, \neg X_n\}$, and a sequence conjunction of clauses $C = (C_1 \wedge C_2 \wedge \cdots \wedge C_m)$, where each clause C_i is a subset of X containing arbitrary literals boolean variable X, and each clause C_i is a disjunction of literals $l_1 \vee l_2 \vee \cdots \vee l_i$. A literal is either a variable X_i or it is negation $\neg X_i$.
Question: Is there a truth assignment to X so that each clause is satisfied?
Answer: "Yes" if for some assignment of boolean values to variables in $\{X_1, X_2, \ldots, X_n\}$ at least one literal in each clause has value 1.

Systematic search or constructive search: System search methods usually organize the search space by partitioning it systematically. This method works systematically by constructing assignments, and make decision about which values to assign to variables. Thereby reasoning about a partial solution is incrementally extended into global solution.

Ternary constraint: All constraints are expressed using relations among 3 variables.

Thrashing: Repeated assignments failure due to the same reason, during search.

Unary constraint: A constraint on a single variable. That is, its constraint arity is equal to one. It is the special case of binary constraint $C_{ij} \subseteq D_j \times D_i$, when $i = j$ denotes a unary constraint on X_i.

Universal constraint: Any combinations of tuples from the variables' domain is allowed.

Variable domain: The set of possible values which variable can take.

Variable-value ordering heuristic: This algorithm is for constraint satisfaction requires the order in which variables are to be considered to be specified as well as the order in which the values are assigned to the variable on backtracking, and it is to consider the order in which variables are considered for instantiations. In this method, the variable with the fewest possible remaining alternatives is selected for instantiation. Thus the order of variable instantiation is, in general, different in different branches of the tree, and it determined dynamically.

Variable-value pair (vvp): The micro-structure graph for a CSP has a vertex for every variable value pair in the original CSP and connects vertices that correspond to compatible assignments in the CSP. It is a combination of two parameters, one is one of the variables from the variable set, and another is one of the value from that variable.

Width of an ordering: Given an undirected Graph $G = (V, E)$, an ordered graph is a pair (G, d), where V =$\{v_1, \ldots, v_n\}$ is the set of nodes, E is set of arcs over V, and d = $(v_1, \ldots, v_n)$ is the ordering of the nodes. The nodes adjacent to v that precede it in the ordering are called its parents. The width of a node in an ordered graph is the number of its parents. The width of an ordering d, denoted $w(d)$, is the maximum width over all nodes.